1995-03

A study of covert channels in a trusted UNIX system

DeJong, Ronald Johannes

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A STUDY OF COVERT CHANNELS IN A TRUSTED UNIX SYSTEM

by

Ronald Johannes DeJong

March 1995

Thesis Advisor: Cynthia E. Irvine
Co-Advisor: Timothy J. Shimeall

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1995	Master's Thesis

4. TITLE AND SUBTITLE
A STUDY OF COVERT CHANNELS IN A TRUSTED UNIX SYSTEM

5. FUNDING NUMBERS

6. AUTHOR(S)
DeJong, Ronald Johannes

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5101

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Analysis and identification of potential channels for illicit information flow is not required for Class B1 trusted systems such as the Sun Microsystems Trusted Solaris 1.1 trusted computing base. When used in a multilevel context such channels would present a risk to data security. The problem addressed by this thesis is the identification of covert channels in Trusted Solaris and the determination if their exploitation can be detected using mechanisms provided to the security adminstrator.

The approach taken to address this problem was to identify covert storage channels in the form of observable effects and exceptions of sharing internal databases by subjects at differing access classes. Software was developed to exploit the identified covert channels using a method requiring detailed specifications prior to the creation of code. Audit trails were obtained to evaluate the efficacy of audit in detecting active covert channel exploitation.

This thesis presents: a list of identified covert channels in Trusted Solaris; a method to circumvent a vendor attempt to close a covert channel; the granularity of auditing required to detect identified covert channels; and an assessment of audit management impact on a posteriori covert channel detection tools.

A major product of this research is a fully operational multilevel security Class B1 TCB system available for further research.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Covert Channel; Audit; Multilevel Secure System;Software Engineering;	101
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

A STUDY OF COVERT CHANNELS IN A TRUSTED UNIX SYSTEM

Ronald Johannes DeJong
Captain, United States Army
B.S., Kansas State University, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1995

Author: [redacted]

Ronald Johannes DeJong

Approved By: [redacted]

Cynthia E. Irvine, Thesis Advisor

[redacted]

Timothy J. Shimeall, Co-Advisor

[redacted]

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Analysis and identification of potential channels for illicit information flow is not required for Class B1 trusted systems such as the Sun Microsystems Trusted Solaris 1.1 trusted computing base. When used in a multilevel context such channels would present a risk to data security. The problem addressed by this thesis is the identification of covert channels in Trusted Solaris and the determination if their exploitation can be detected using mechanisms provided to the security adminstrator.

The approach taken to address this problem was to identify covert storage channels in the form of observable effects and exceptions of sharing internal databases by subjects at differing access classes. Software was developed to exploit the identified covert channels using a method requiring detailed specifications prior to the creation of code. Audit trails were obtained to evaluate the efficacy of audit in detecting active covert channel exploitation.

This thesis presents: a list of identified covert channels in Trusted Solaris; a method to circumvent a vendor attempt to close a covert channel; the granularity of auditing required to detect identified covert channels; and an assessment of audit management impact on a posteriori covert channel detection tools.

A major product of this research is a fully operational multilevel security Class B1 TCB system available for further research.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

I want to thank Prof. Irvine for igniting my interest in computer security, and for her guidance and patience during this research.

Albert Wong, the Research and Development Manager of the Computer Science Department, was instrumental in the successful configuration and installation of the trusted multilevel system. His many years of experience in distributed system administration was an ever-ending source of information. Not only was it a pleasure to work with and learn from him, but it was fun. My only regret is that I will not be here longer to tap his knowledge in system administration. Thanks Al!

I would also like to extend a thank you to the following individuals for their time, patience, and energy in helping make this thesis materialize:

- Patrick Barrett, Sales Representative, Sun Microsystems Computer Corporation
- Prof. Harold Fredricksen, Mathematics Department,
- Prof. Frank Kelbe, Naval Postgraduate School
- Prof. Timothy Shimeall, Naval Postgraduate School
- Prof. Man-Tak Shing, Naval Postgraduate School
- Prof. Roger Stemp, Naval Postgraduate School
- Sue Whalen, Lead Systems Administrator, Naval Postgraduate School
- Mike Williams, Lead Hardware Technician, Naval Postgraduate School

A very special thank you to my wife Linda for her unwavering love and support which allowed me to successfully complete my graduate education.

I. INTRODUCTION

A. PURPOSE

The Department of Defense (DoD) agencies are directed to purchase Commercial Off the Shelf (COTS) software and systems that will meet automation requirements when available. The identification of potential channels for illicit information flow is often absent in the documentation of these systems due to the vendors' lack of knowledge or reluctance to publish them. A number of relatively small channels for illicit information flow may exist in a multilevel secure system. The existence of these illicit channels is usually buried deep in the Trusted Computer Base (TCB) system design, and it may be impractical to eliminate them all because of cost or the effects on overall system performance. Although it is possible to audit the use of these small illicit channels without adversely affecting the system performance, the auditing may not always be enough to assure their detection due to the vast amount of information generated during an audit trail. The Information System Security Officer (ISSO) may never detect an information security breech during human analysis of the audit trail, unless they are aware of what entries to look for. Without automated auditing tools humans can easily overlook covert information flows. [Ref. 1]

The major thrust of this thesis involved the installation (and subsequent administrative management) of a multilevel secure network operating with Sun Microsystems Computer Corporation's *Trusted Solaris 1.1* TCB. This provided a fully operational TCSEC Class B1 TCB system for further study and experimentation on problems involving multilevel security. A number of illicit information flow channels were identified in the system, and a mechanism was created to exploit them. Additionally, the research shows that the ability of a system to audit a covert channel is, in itself, insufficient to detect its use.

B. BACKGROUND

The computer security problem has increased with the growth of computer usage. The problem of security was not really noticed in the early computers because the size and nature of their applications could be resolved in the environment surrounding the computer system. Systems that processed sensitive information were merely locked in a room; granting access to authorized users only. However, users began demanding better utilization of computer resources around the mid-sixties. This gave birth to timesharing computer systems that served multiple users simultaneously. With the use of timesharing systems came the problem of controlling the processing environment and data protection. In the late 1960's the DoD began development of a way to protect classified information stored on computers. Prior to that time it was against DoD policy to process classified information on a computer system that uncleared individuals had access to [Ref. 2, p. 63]. As the use of computers further increased, the DoD put forth effort to develop *multilevel secure systems* that could protect multiple levels of classified information on a single computer system and enforce the associated security policy; in essence, control the sharing of resources.

This introduced a new twist to the disclosure-of-information problem, known as *illicit information flow.* Illicit information flow involves the dissemination of information to an unauthorized user or process. Lampson [Ref. 4] was the first to address the problem of illicit communication through legitimate mechanisms in 1973, and discussed confining a program during its execution so that it can only transmit information back to its caller. An illicit activity that normally took hours in a "paper" environment, could now be accomplished in seconds on a computer, due to the enormous storage capability and ever decreasing access times. Over the past two decades, the computer security community has placed a great deal of attention on the disclosure of information threat. Two significant mathematical information flow models were developed during this period; known as the *Bell and La Padula model* and the *Biba integrity model.* [Ref. 3, p. 4]

Bell and LaPadula (BLP) [Ref. 4] devised a security model that has had an enormous impact on research and development in computer security. The security policy reflected in the BLP model has two components comprised of discretionary and mandatory

access control rules. The discretionary component can be represented in an access control matrix model similar to the Graham-Denning model [Ref. 5], and has about 20 functions for dealing with modifying components of the matrix. The mandatory component consists of the simple security property, also known as the *no read up* rule, and the confinement property (*-property), also known as the *no write down* rule. Both properties enforce mandatory access control by restricting accesses based on a comparison of access class attributes (a combination of access modes such as read, write, and execute) of *subjects* and *objects* on the computer system. An object is anything that holds data, such as a file or memory, and a subject is anything that can access or manipulate objects, such as a process. A user is viewed as a subject at the highest level of abstraction from the system.

The BLP model only dealt with the secrecy or disclosure of information, and failed to control unauthorized modification of information, also known as *data integrity.* In this context, data integrity means that information is modified only by those having the right to do so. In the mid 1970s, Biba [Ref. 6] made this observation about the BLP model and developed an integrity model that is nearly identical to the BLP model, except that the rules are reversed. The simple integrity rule is *no read down*, and the confinement property (*-property) is *no write up*. Both these rules enforce integrity by restricting accesses based on a comparison of integrity access class attributes of subjects and objects.

Mandatory access controls can only prevent illicit communication across access classes. However, computer systems are designed with various *overt* (legitimate) mechanisms intended for providing interprocess information and services communication. Examples of such mechanisms are a file, an interprocess message, shared memory, and system performance information. When a computer system mechanism can be used in an unexpected manner to disclose information to an unauthorized individual, then a *covert channel* exists.

A covert channel is defined as: "any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy." [Ref. 1, p. 81] The two types of covert channels are *storage* and *timing*. A covert storage channel is any communication path that results when one process directly or indirectly

writes to a storage location that another process observes by the direct or indirect reading of the same storage location. The covert storage channel can be further subdivided depending on which of the following three types of information it uses: object attributes, object existence (or nonexistence), and shared resources. A covert timing channel is any communication path that results from one process modulating its own use of system resources, which another process observes by measuring the changes in response times for its use of the same system resources.

A *Trojan Horse* can be written to take advantage of a covert channel, and is an important concept because it eliminates the need for cooperation between two individuals for the exchange of illicit information. Any program that appears to the user to perform some desirable (overt) function, but in fact carries out some illicit or undesirable (covert) function is a Trojan Horse. Thus, a Trojan Horse program replaces an existing program on the system that is invoked by an unsuspecting user. The Trojan Horse appears to function as normal to the user, but is actually exploiting some covert channel or other malicious actions in the background. Consider the following scenario, a user invokes the "vi" edit command on a UNIX system that has been corrupted by a Trojan Horse. The "vi" editor appears to function normally to the user, but is actually sending each character you type in to an unauthorized user's file by exploiting a covert storage channel. The storage channel exploits the fact that other users can monitor file size information (object attributes). First, the Trojan Horse creates a file (or uses an existing file) in the users current directory. Then for each character typed in, the file is set to a specific size. A process being run by the unauthorized user monitors the file size changes; which converts the size to a character using a predetermined mapping function. Finally, when the legitimate user ends the "vi" editing session the Trojan Horse signals an "end of transmission" to the unauthorized user's process before shutting down. This is only one of many scenarios that prompted user to demand more secure systems.

The National Computer Security Center (NCSC) was formed in January 1981, and given the major goal "to encourage the widespread availability of trusted computer systems for use by those who process classified or other sensitive information." [Ref. 1, pp. 1-2] In

1983 the NCSC published the *Trusted Computer System Evaluation Criteria* (TCSEC) [Ref. 1], also known as the "Orange Book." It was revised in 1985 and adopted by the DoD as the standard criteria for trusted computer system evaluation. This document, commonly referred to as TCSEC, provides standards and guidelines to manufacturers on how to build and evaluate systems built for commercial or military applications. The TCSEC provides seven evaluation criteria classes ranging from systems that have minimal protection features (D) to those that provide the highest level (A1) of *assurance* that the system enforces the following requirements: the security policy, marking of storage objects with access control labels, identification of individual subjects, and accountability of security related actions (auditing). The NCSC has since published a set of trusted system TCSEC interpretations and guidelines, referred to as "the Rainbow Series," that further assist in the development and usage of trusted computer systems.

The TCSEC requires covert channel analysis (CCA) for systems in classes B2 through A1. Although CCA is not required for TCSEC class B1 systems, the potential risk exists that the system may be used in an operating environment with a wider range of TCSEC class systems. The TCSEC class B1 system may also be used by individuals, or to process information, outside the approved range of clearance or data sensitivity levels. It is for these reasons that the following analysis has been conducted.

C. RESEARCH OVERVIEW

In this section an overview is given of each chapters contents.

1. Introduction

2. System Setup and Configuration

This chapter is not intended as a stand alone instructional manual to installing a trusted computer base (TCB). It provides highlights of the preparation and planning, and contains an overview of the installation process and system software and hardware configurations. Further, it gives observations noted during the installation of Trusted Solaris that generally hold true for the installation of any trusted system. The most

significant observation is that all members of the installation team should have prior experience in administering a network.

3. Covert Channels

Chapter III contains the descriptions of the covert channels discovered on the Trusted Solaris 1.1 system. Additionally, an explanation is given of the *information theory* technique used to encode the data passed by the covert channels. Appendix B contains the specifications for the covert channels exploitation code. The actual exploitation of the discovered covert channels does not synchronize between sender and receiver to ensure efficient transfer of information. However, this research shows that covert channels exists and are in fact exploitable.

4. Software Engineering

Chapter IV discusses the importance of a disciplined approach to software development. The *goals* and *principles* of software engineering techniques to properly decompose a software system into modules are summarized. Further, a description of the method used to develop the process specifications contained in Appendix B is given. The process specifications represent the largest amount of detailed work in building a software system. Although the covert channel software system is small enough that it could have been designed without following all the rigors of software engineering, it was done as an exercise to fully appreciate and understand the methods used to develop a TCB system. However, the observations made during the development of this relatively small system, with respect to the value of the software engineering process, proved to result in a system that is easily maintainable and more efficient.

5. Audit Data Collection

This chapter describes the auditing process and discusses the TCSEC requirements for auditing in a trusted system. It further provides an overview of how auditing is configured in Trusted Solaris 1.1, and the auditing tools available to the ISSO for managing and analyzing the audit information generated by the audit mechanism.

6. Analysis

Chapter VI provides an analysis of the auditing tools available to the ISSO on Trusted Solaris 1.1, implications of auditing on system performance, and a discussion of the discovered covert channels. The analysis demonstrates the need for an additional auditing application to assist the ISSO in managing and analyzing the enormous amount of auditing information generated. Additionally, it shows that there is no easy "fix" to close the discovered covert channels.

7. Conclusion

Chapter VII provides a summary of the initial research findings and offers recommendations for future research.

II. SYSTEM SET UP AND CONFIGURATION

A. PREPARATION

1. UNIX System Administration Tutorials

The Trusted Solaris 1.1 kernel is UNIX-based. Having no prior knowledge or experience in network administration and limited expertise in the UNIX environment, attending the series of tutorials on UNIX system administration was extremely helpful. The two part series of tutorials was offered by the UNIX System Support Committee and taught by experienced system administrators at NPS. Part one of the series covered the intricacies of the UNIX operating system from a system administrator's perspective and provided a thorough understanding of how the system works. Part two discussed the concepts of networking, client/server operations, and network services.

2. Hardware

The hardware portion of the trusted computer system consisted of three interconnected *SPARCstation 10* (sun4m) workstations, an external tape drive, an external CD-ROM drive, and a Hewlett Packard LaserJet III printer. Two workstations had one 405MB internal hard drive each, and the other had two 405MB internal hard drives.The configuration of the system is discussed in section B.2 of this chapter.

3. Software

An evaluation copy of *Trusted Solaris 1.1* from the SUN Microsystems Computer Corporation was used. Trusted Solaris 1.1 is a TCSEC class B1 system [Ref. 1, pp. 20-25].

B. PRE-INSTALLATION

This portion of the installation was the most time consuming. All the *Trusted Solaris* manuals needed to be reviewed and an in-depth understanding was required of the contents in the *Security Features Users' Guide* and the *Trusted Facility Management Manual* volumes I and II. Only then could the planning of the system configuration start.

1. Installation Team

In *Trusted Solaris,* the traditional UNIX "superuser" role and its associated tasks and capabilities are distributed among the following three default trusted facility management roles: Information System Security Officer (ISSO), system administrator, and system operator. The ISSO performs all the security-relevant administrative tasks for the trusted system, such as configuring the other roles, user clearances, setting up object labels, and managing and analyzing the audit data (discussed in Chapter V). The standard non-security-relevant UNIX system administrative tasks are performed by the system administrator. The system operator is responsible for conducting system backups and machine reboots.

This separation of duties is a major aspect of trusted facility management and is part of the principle of *least privilege,* which requires that system managers be given no more access than is needed to do their job [Ref. 2, p. 49]. The separation of management tasks reduces the security risks resulting from human error, deliberate wrong doing, and system failure. The ISSO and the system administrator must work together to plan and install the system; the system operator is only needed once the system is up and running.

The ISSO role is by far the most critical position. The ISSO must plan the installation to be consistent with the Site Security Policy. Therefore, the ISSO must:

- Understand Site Security requirements.
- Thoroughly understand the workings of the trusted system in order to set up and maintain the safeguards that protect the security of the information in accordance with the Site Security Policy.
- Have a clearance equal to or greater than the highest security level of information processed on the system.

The ISSO and system administrator should already have experience administering SunOS on Sun machines, and together they must learn and understand the differences between SunOS and Trusted Solaris. Sun recommends a senior system administrator assist the ISSO in the software installation. Together, these installation team members should already have an understanding of: local network configuration, server support services, SUN hardware configuration, and local site security requirements. [Ref. 5, pp. 138-140]

2. Planning the Configuration and Installation of the Distributed System

In order to preserve the trusted computing base (TCB), the closed net distributed system was configured with a network file system (NFS) and network information services (NIS) server and two dataless clients. A dataless client is a machine that has its own copy of the Trusted Solaris operating system on the local disk drive, but because it receives (mounts) the */home* and */etc/security/audit/<server name>* file systems from the NFS server; it cannot completely boot without the NFS, NIS, and Audit servers. The */home* NFS contains all the users' working directories and the *.../audit/<server name>* is where all the audit data is written.

3. Gathering Preliminary Information

The *Trusted Facility Management Manual* [Ref. 5] contained excellent worksheets for planning the installation of Trusted Solaris on each machine. These worksheets covered all practical aspects of the installation such as: network information, software categories to install, overall disk partitioning, and auditing.

C. INSTALLATION AND CONFIGURATION

The actual installation required minimal time. This is mainly attributable to the amount of prior planning and experience of the senior administrator. The installation is predominantly menu driven using the *suninstall* program. However, the only type of server that is fully configured during *suninstall* is the diskless client server. This required the additional exporting of the following file systems from the NFS server to the clients: */usr* (sd0), */home* (sd1), and */usr/security/audit* (sd1). Although this is easily within the expertise of the senior administrator, the ISSO must also have a basic understanding to insure the integrity of the system is maintained and that the entire configuration supports the site's security policy.

A Hewlett Packard LaserJet III printer was later added to the system. This printer is not accredited for the Trusted Solaris system and causes an *extended* configuration of the

TCB. Although the printer was not used in conjunction with any covert channel exploitation, its effect on assurance protection is unknown.

D. SYSTEM BACKUP

This aspect was especially important because the entire installation could not be accomplished in one "sitting" due to other commitments of the install team members. Therefore, a backup was made at the end of each session of any file, or file system, altered during the configuration process; thus preventing loss of data and valuable time. Although backups are performed the same as for a SunOS system, the backup media requires handling commensurate with sensitivity level of the data processed on the system.

E. SUMMARY

The following listing is a summary of the major steps involved in the TCB system configuration and Trusted Solaris 1.1 software installation:

- Installation team members thoroughly review system manuals; noting differences between Trusted Solaris and SunOS.
- ISSO determines how the system must be configured to meet the Site Security Policy.
- ISSO and senior system administrator plan the configuration of the distributed system (server and client types).
- Gather preliminary information for the installation by filling out the provided worksheets.
- Partition the internal disks of each machine according to the sizes on the worksheets.
- Install and configure the master server (NIS/NFS).
- Install and configure the client machines.
- ISSO and senior system administrator configure the trusted system management roles.
- ISSO configures auditing on all machines.
- ISSO turns auditing and TCB verification "on" in the */etc/rc.local* file of all machines. Then synchronizes (updates) the TCB verification databases on all machines. The TCB verification satisfies the "System Integrity" requirement of the TCSEC. [Ref. 1, p. 24]

- Reboot the server, and then reboot the clients.

F. COMMENTS

The following comments are observations noted by the installation team during the process of installing Trusted Solaris that generally hold true for the installation of any trusted system:

- The amount of time and effort placed on the preparation and planning phase by the installation team is instrumental in the successful installation of the system.
- Maintain a log/journal during the installation to insure completeness. This is especially helpful when the installation cannot be completed in one sitting.
- Leaving *auditing* off and not synchronizing the TCB consistency databases until the entire system was configured saved time. The installation manual has the install team synchronizing the TCB consistency databases multiple times in different sections of the installation process. The command to synchronize the TCB consistency databases after each particular NIS file is properly altered is cumbersome and time consuming. Synchronizing the TCB consistency databases after all files have been altered and then re-booting does not affect the proper installation of the system. However, this should only be done in a secure site environment with controlled access. This insures that no single member of the install team having knowledge of the *boot* password can tamper with the system.
- Obtain errata sheets for all manuals. Many instructional errors were found that caused unneeded delays.
- The system is only as good as the trusted facility management personnel. If the trusted facility management personnel do not have the qualifications and knowledge discussed in section B.1, the system may not be configured or managed properly. This will inevitably lead to an untrusted system.

An excellent overview of the Trusted Solaris 1.1 operating environment is provided in Appendix A. The reader is highly encouraged to scan the overview to gain a basic understanding of how the system works and how to work with the system.

III. COVERT CHANNELS

This chapter contains the descriptions of the covert channels investigated on the Trusted Solaris 1.1 system. Explanations of the method used to investigate the existence of covert channels on the system and the *information theory* technique used to encode the data passed by the covert channels are provided. Appendix B contains the specifications for one covert channel's exploitation code, and Chapter IV discusses the software engineering techniques used to develop the process specification. An analysis of the covert channels identified is provided in Chapter VI.

This research does not synchronize the transfer of information between the sender and receiver processes. That is, some information is lost in the transfer due to the processes running at different speeds and do not synchronize ("shake hands") after each piece of data is exchanged. However, the research shows that covert channels exists and can in fact be exploited in the system.

A. METHOD TO IDENTIFY COVERT CHANNELS

The NCSC *A Guide to Understanding Covert Channel Analysis of Trusted Systems* [Ref. 18] gives three primary sources of covert channel identification:

- System reference manuals containing descriptions of TCB primitives, CPU and I/O processor instructions, their effects on system objects and registers, TCB parameters or instruction fields, and so on;
- The detailed top-level specification (DTLS) for B2-A1 systems, and the Formal top-level specification (FTLS) for all A1 systems; and
- TCB source code and processor-instruction (micro) code.

Only the first source was available for this research, and there are certain disadvantages when having to strictly rely on system manuals for covert channel identification. First, the TCB and processes can only be viewed as black boxes; thus, hiding the details of the system. Only guesses and possible analogies with specifications of other systems known to contain covert channels can be used as a means for identification. Therefore, all covert channels may not be identified. Second, "few identification methods

exist that exhibit any degree of precision and that can rely exclusively on information from system reference manuals." [Ref. 18]

The method used to investigate the existence of covert channels in the Trusted Solaris system involved a bottom-up systematic approach to finding kernel or system call databases that might share information between processes at different sensitivity levels. First, the UNIX header (*.h*) files were examined to identify databases that might be shared by processes at different access classes. Then all system call commands listed in the manual were intuitively scanned for possible usage of the databases identified. Finally, the identified system call commands were used simultaneously at different sensitivity levels to determine if the database information was being shared. This identification method led to the file system modulation covert channel discussed in Section C. One particular database identified, that is not sharable, is the process database. A user or process only has access to information about processes at its own sensitivity level.

In addition to searching for shared databases, some shared system resources were explored. This led to the identification of two additional covert channels: memory and device allocation. These covert channels are further discussed in Sections D.1 and D.2.

B. INFORMATION THEORY METHOD

The purpose behind the exploitation of covert channels is the illicit transfer of information from a high to a low security classification level. An example of such information might be a military Battle Plan or sensitive corporate information. In a multilevel system this information is considered secure (protected from view by lower classified users) and is normally stored in plain text form.

The study of information theory regarding the percentage of characters needed from a document in order to extract the meaning of its contents, and the numerous methods of encoding the data, is beyond the scope of this thesis. However, a simple (yet effective) cryptographic technique was used which involves encoding only the alphabetic and numeric characters. All uppercase alphabetic characters are converted to lowercase prior to

encoding. The probability distribution of alphabetic characters in the English language [Ref. 9] was then used to map the letters to a rank ordered number code based on the probability. For example the letter "e" occurs with the highest probability and the letter "z" with the lowest, and they would be mapped to zero (0) and twenty-five (25) respectively. The numeric characters zero through nine (0-9) are mapped to 26 through 35. Giving a total of 36 character codes (0-35).

The codes can then be transmitted by whatever means the covert channel employs. Although one of the exploited covert channels allows the full transmission of the character code, most other covert channels only allow a single bit (flag) to be set or unset. In such instances the character code requires further encoding to a binary signal by converting the 36 character codes to binary using groups of six bits ($2^6 = 64$).

C. FILE SYSTEM MODULATION COVERT CHANNEL

1. Description

This covert channel exploits the fact that files are stored in a *file system*. A file system corresponds to a finite area (partition) of disk. The amount of space available and allocated to each file system is printed by the *df* command or by making the direct system call to *statfs* from within a process. The disk space on the file system is allocated in units called blocks; a block is typically 512 or 1024 characters, the latter being the case of Trusted Solaris. The *statfs* system call receives the information from a system database structure called *fs*, the details of which can be found in the */usr/kvm/sys/ufs/fs.h* file. This command, and subsequent file system information, is available to all users and is especially useful to ensure that enough space is available prior to creating a large file.

Consequently, this system mechanism can be exploited by a low classified user observing file system size changes that another higher classified user is making. The high user may intentionally modulate the file system size to signal the low user, or the low user can implant a *Trojan Horse* to accomplish the task. The Trojan Horse can start up when the high user opens a file for editing, and will covertly send the file contents to the low user.

The following scenario describes how the file system information is exploited. The high (i.e.; top secret) classified user implements a program that opens the file containing the information to be sent to the low (i.e.; unclassified) user. The previously mentioned information theory cryptography technique is used to encode the file contents (characters), and the size of another file is changed to the number of blocks corresponding to each character code. The file size is increased or decreased by the *truncate* system call according to the number of blocks given as the parameter. Simultaneously, the low user starts a program that opens a file for writing and makes an initial *statfs* system call to establish a base file system size. The number of *free blocks* remaining is a statistic returned from *statfs*, and is used to monitor the file system size. Subsequent calls to *statfs* compare the current number of free blocks to the base; the absolute difference of the two is one encrypted character code, which is mapped back to a character by reversing the same cryptography technique used by the high user. This process continues until a prearranged end of file signal is received.

2. Attempted Vendor Fix

First, all *statfs* and *truncate* system calls are able to be audited. However, discovering the exploitation of the covert channel through the audit trail can be difficult, to say the least. This topic is discussed in greater detail in Chapter V.

The second "fix" is to somehow monitor file system size changes caused by the *truncate* system call to alter file sizes, and then return random file system size information from a *statfs* system call. All information is accurate for a file beginning with a size of zero and increased in size up to eight blocks (one block = 1024 bytes = Kbyte). After eight blocks the *statfs* information is erroneous and unpredictable.

3. Workaround

Further analysis revealed the following two properties whereby the *statfs* system call returns accurate information. First, it was discovered that multiple files can be set to sizes ranging between zero and eight blocks. Additionally, if a file is set to any size up to

eight blocks, it can then be increased an additional eight blocks if the file size is not returned to zero each time.

These discoveries lead to two important techniques employed in the exploitation of the file system covert channel. First, the original encryption technique is possible by using five files for modulating the file system size; thus giving a total file system size modulation range of 41 blocks (0 - 40, or 0 - 8 times five files). Second, a base of 10 blocks (two files always set to a minimum file size of five blocks) is possible due to the discovery that any file can be increased in size an additional eight blocks if not returned to a block size of zero. The "base" can then be used as an end of transmission (EOT) signal by setting all file sizes (including the "base" files) to zero; thus, showing the low user a negative file system size change.

D. ADDITIONAL COVERT CHANNELS IDENTIFIED

The following two covert channels require that two processes, of different sensitivity levels, run simultaneously on the same machine. This is accomplished by logging in as a high sensitivity level user and opening two command tool windows; one at a low sensitivity level and the other at a higher sensitivity level. Commands or processes can then be initiated from within each window. This is similar to the two users on separate machines for the file system modulation covert channel; in the current case, the users are two separate command tool windows, at different sensitivity levels, on the same machine.

1. Memory Allocation

This covert channel exploits the fact that a finite amount of memory area is shared by processes. If one process allocates a major portion (over half) of the memory area, then another process is unable to allocate a similar amount and receives an error that the memory space has not been allocated. For example, if 30 megabytes of memory is available and process "A" allocates 20 megabytes, then process "B" will receive an error when it tries to allocate the same amount. Unfortunately to reduce covert channel exploitation, the system locks all memory area acquired by a process; even though the process deallocates (frees)

the memory space, another process is still unable to use any of the locked memory area until the process that allocated the area terminates. However, a signal can be sent between processes using this method.

2. Device Allocation / Deallocation

This covert channel exploits the fact that devices, such as floppy and tape drives, can only be allocated to one process at a time. This covert channel is similar to the memory allocation method, but uses the error that a device has already been allocated.

IV. SOFTWARE ENGINEERING

A. PURPOSE

A disciplined approach to software development, in the form of proper software development methods, ensures the *correctness* of software systems. To ensure correctness in a trusted computing base (TCB) system, it must be "internally structured into well-defined largely independent modules" [Ref. 1, p. 30]. In the case of illegitimate software, its developers must be confident that the Trojan Horse acts correctly in order to avoid detection. The following method was used in the development of the covert channel software, and the resulting process specifications are provided in Appendix B.

B. GOALS OF SOFTWARE ENGINEERING

The underlying goal in software development is that the resulting system meets the stated requirements. However, the requirements will inevitably change over the life cycle of the system. The following four general properties are accepted as *goals* for the overall software engineering discipline, and aid in the transition between changes. [Ref. 10, pp. 18-20]

1. Modifiability

A software system may require modifications to correct an error, or in response to changes in the requirements. To effectively modify the system, the original design decisions must be maintained across changes or the original structure will be obscure and further complicate future modifications to the system. Although the modifiability of a software system is hard to measure, the goal is to be able to introduce changes without increasing the complexity of the original system.

2. Efficiency

This goal implies that a software system should operate using the available time and space resources in an optimal manner. Time resources are mainly dependent on the hardware architecture, but the choice of software algorithms will impact the overall

execution time. Space resources refer to the physical considerations of the system such as memory and peripheral devices. The software system must consider both classes of resources during design. One fault is to concentrate on microefficiency of the resources in the early development phase instead of macroefficiency. An early understanding of the overall problem will yield more efficiency than the maximization of each resource throughout the design, without regard for other processes.

3. Reliability

A computer system is expected to be reliable in the prevention of or recovery from failure. However, it must also provide reliability in performance and the correctness of operations. For instance, the TCB must be reliable in the enforcement of mandatory and discretionary access control to sensitive information or processes.

4. Understandability

This goal is the most critical in managing the complexity of the software system. Every level of the development phase should ensure understandability. From the mapping of the objects and operations to the requirements at the highest level, to the readability of the code at the lowest level. Since the development of any software system is bound to have a certain degree of personnel turnover, understandability of the system leads to earlier productivity from new personnel.

C. PRINCIPLES OF SOFTWARE ENGINEERING

The following are principles that will assist in developing software systems that meet the above goals of modifiability, efficiency, reliability, and understandability. [Ref. 10, pp. 20-24]

1. Abstraction and Information Hiding

One of the fundamental principles for managing the complexity of the software system solution is *abstraction*. The objective of abstraction is to extract the basic elements, yet omit the details of the elements. The elements of the solution are the algorithms, and their associated data. Each element in the decomposition becomes a part of the abstraction

at a certain level. Figure 1 shows an example of how abstraction can be applied to file access elements.

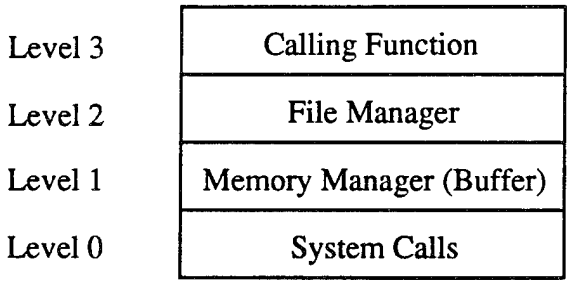


Figure 1. Example of abstraction levels for file access.

Closely related to abstraction is the principle of *information hiding*. Although the objective of abstraction is to extract the details of a given level, information hiding makes the details of a particular element inaccessible to other elements. By hiding the details of one level from another, the cascading of changes can be prevented.

The principles of abstraction and information hiding reduces the details a developer at one level needs to know about the other level; therefore, aiding in the goals of modifiability and understandability of the overall software system. The software engineering principles also aid in the goal of readability by localizing an operation to one element in a level and preventing the dependency of the operation on other elements.

2. Modularity and Localization

Another principle that helps in managing the complexity of a software system is *modularity*. The general decomposition techniques are *top-down* and *bottom-up* design. With top-down design, each successive level of the system is decomposed into modules; with the properties that higher level modules specify what is to be done, and the lower level modules specify how the action is to take place. Whereas, bottom-up design employs the same properties except that the system is built from low level modules to ever increasing complex ones. A large complex system will normally employ both design methods [Ref. 10, p.23]. Decomposing from the top down and then building from the bottom up with existing reusable modules.

The *localization* principle assists in creating modules that exhibit loose *coupling* and strong *cohesion*. Coupling is a measure of the degree to which modules are interconnected. A loosely coupled module is relatively independent of others. Cohesion is the measure of how tightly bound the internal elements of the module are to each other. A strongly cohesive module has functionally and logically dependent components.

The conventional method for modularization, decomposes the software system using a flowchart and then assign one or more subroutines to each module. A more unconventional method is proposed by Parnas for more complex systems; usually beyond the order of 10,000 instructions. First, the difficult and likely to change design decisions are listed. Then modules are designed based on the decisions. The reasoning behind his method is, "Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing." [Ref. 11]

Both conventional and unconventional methods are expected to support the same goals of software engineering: modifiability, reliability, and understandability. A well modularized structured system limiting the coupling between other modules will be easier to understand and increase reliability. Additionally, the localization of design decisions will prevent the cascading of modifications across modules.

3. Uniformity, Completeness, and Confirmability

These final principles ensure that a software system will be consistent and correct. *Uniformity* requires that a consistent control structure and calling sequence for related objects is the same at level used, and is normally achievable through good coding styles. *Completeness* ensures that only necessary and important elements are present in the design. Finally, *confirmability* ensures that the system has been properly decomposed so that it can be readily tested. The confirmability principle is especially imperative for a TCB system design, since "Testing shall demonstrate that the TCB implementation is consistent with the descriptive top-level specification." [Ref. 1, p. 31]

All four software engineering goals are supported by these principles. Uniformity directly supports understandability by using consistent notation. Modifiability, efficiency,

and reliability are supported by the principles of completeness and confirmability through the development of correct (testable) solutions.

D. SPECIFICATIONS

The underlying purpose of the previous sections is to properly decompose the software system into modules. This section covers the aspects of capturing the behavior and structure of the software system with relevance to the development team. "An architectural design describes the internal and external interfaces along with the concepts needed to build the proposed system." [Ref. 12, p. 207]

1. Initial Specification

The process starts by considering a major module of the system decomposed by the previous methods as a single *black box* (localization of module), and proceeds to decomposing the black boxes into more primitive ones, until all the primitive boxes are simple enough to be implemented by a single function or procedure of code. There is no current method for conducting the black box decomposition on a computer. However, Berzin and Luqi [Ref. 12] recommend using the following guidelines for human designers:

- Sketch an informal algorithm for the module, limited to about 10 lines. The limit of 10 lines for the informal algorithm can be relaxed for regular structures such as case statements with many independent cases at the same level of abstraction, provided that the actions to be performed in each case are encapsulated in lower-level modules.
- Identify the primitives used that are not already available, and specify them as modules using the techniques for functional specification.
- Repeat the process for the new primitives introduced in the previous step, if there are any.

An important decision is whether or not to decompose a primitive black box further. The basic guideline is that each primitive should be small enough to code and compile with about one person-day of effort, not counting design and testing. [Ref. 12, p. 215]

There are many tools and formats that can be used to produce a process specification, such as: flowcharts, decision tables, structured English, pre/post conditions,

and diagrams. Although structured English is favored by most system analysts, Yourdon [Ref. 13, p. 203] gives two crucial requirements that any method used must meet:

> • *The process specification must be expressed in a form that can be verified by the user and the systems analyst.* It is precisely for this reason that we avoid narrative English as a specification tool: it is notoriously ambiguous...
>
> • *The process specification must be expressed in a form that can be effectively communicated to the various audiences involved.* While it will typically be the systems analyst who *writes* the process specification, it will usually be a diverse audience of users, managers, auditors, quality assurance personnel... who read it.

MODULE **CONSTANTS:** Globals to all the functions in the module.
TYPES:
VARIABLES:
DATABASES:
Databases/structures internal to the module.
FUNCTION DESCRIPTION:
Text description of the function operations.
INPUTS
"In" type parameters to the function.
PROCESSING
Structured English.
EFFECTS
Resulting effects of the function on global structures and variables.
EXCEPTIONS
Exceptions resulting from user input or system calls.
OUTPUTS
"Out or InOut" type parameters and functions returns.
DATABASE REFERENCED
Global databases/structures accessed by the function.
INTERFACE: "C" language specific syntax for use of function.

Figure 2. Example of process specification format.

An example of the format used for the development of the covert channels process specifications is given in Figure 2. Structured English was used to describe the processing

for each function. The interested reader can find a full listing of the process specifications developed for the covert channels.

2. Reviews

There are normally three different kind of reviews: customer, internal, and reviews by domain experts. The purpose of the customer review is to ensure the software system is meeting the requirements of the original problem. The purpose of the internal review is to find feasibility, performance, or cost problems, and to find inconsistencies in the requirements. The purpose of the domain expert review is to find problems in the application domain that have not been addressed in the requirements. [Ref. 12, p. 67]

3. Maintenance

Once the development of the software system is complete, at some point during the life cycle there will be changes or additions to the requirements of the system; requiring a corresponding change to the specification and software. Some or all of the development team personnel may no longer be associated with the project. These modifications will require less effort by the current development team if the previously discussed goals and principles of software engineering were followed. However, to ensure the ease of future changes, each subsequent change should adhere to the software engineering goals and principles. Additionally, all changes should be well documented and a copy of each version of the software specification maintained. Many software companies use automated applications to maintain the software system's documentation over its life cycle.

E. SUMMARY

The process specifications represent the largest amount of detailed work in building a software system. Although the covert channel system is small enough that it could have been designed without following all the rigors of software engineering, it was done as an exercise to fully appreciate and understand the methods used to develop a trusted system. However, the following observations were made during the development of this relatively small system with respect to the value of the software engineering process:

- Internal reviews of the process specifications eliminated most logical errors prior to coding.
- The application of the software principles during the system specification development resulted in more compressed and efficient code. Specifically, the function code written to test the feasibility of the file size modulation by the high user was improved (Appendix B, Sec. G.4.2). Not only was it made more efficient by eliminating over half of the conditional statements, but it also became more readable.
- Testing and debugging of the modules was easier due to the principles of software engineering applied. Each function in a module could be evaluated for correctness.

V. AUDIT DATA COLLECTION

A trusted computer system must provide authorized personnel with the ability to audit any action that can potentially cause access to, generation of, or effect the release of classified of sensitive information. The audit data will be selectively acquired based on the auditing needs of a particular installation and/or application. However, there must be sufficient granularity in the audit data to support tracing the auditable events to a specific individual who has taken the actions or on whose behalf the actions were taken. [Ref. 1]

A. PURPOSE

The auditing in a secure system is the process of recording, examining, and reviewing any or all security-relevant activities on the system, and is required for TCSEC [Ref. 1] Classes C2 and above. Users, and their processes, can be monitored for attempts to compromise the security of the system, and analysis of the audit can help determine the extent that the system has been penetrated. The device used to detect and collect all the security-relevant activities is called the *audit mechanism*, and has five important security goals: [Ref. 14][Ref. 15]

- must allow the **review** of patterns of access to individual objects, access histories of specific processes and individuals, and the use of the various protection mechanisms supported by the system and their effectiveness.
- must allow **discovery** of both users' and outsiders' repeated **attempts to bypass** the protection mechanisms.
- must allow discovery of any **use of privileges** that may occur when a user assumes a functionality with privileges greater than his or her own, i.e., programmer to administrator.
- must act as a **deterrent** against perpetrators' habitual attempts to bypass the system protection mechanisms. However, to act as a deterrent, the perpetrator must be aware of the audit mechanism's existence and its active use to detect any attempts to bypass system protection mechanisms.
- supply an additional form of **user assurance** that attempts to bypass the protection mechanisms are recorded and discovered. Even if the attempt to bypass the protection mechanism is successful, the audit trail will still provide assurance by its ability to aid in assessing the damage done by the violation, thus improving the system's ability to control the damage.

Users of the audit mechanism fall into two categories. The first consists of the auditor, normally the Information System Security Officer (ISSO), who selects the system events to be audited, enables the auditing conditions (flags) to record those events, and analyzes the audit trail produced by the audit mechanism. The second category consists of all the system users: administrators, operators, programmers, and all others. Although this category of users does not have direct access to the audit mechanism, they are considered users because they generate audit events. Additionally, they must know the existence of the audit mechanism and what impact it has on them; otherwise, the user deterrence and assurance security goals of the audit mechanism cannot be met.

The product generated by the audit mechanism, in accordance with the chosen audit events, is the *audit trail*. The TCB should provide the auditor (ISSO) with a *pre-selection* mechanism to indicate which auditable events will be recorded on the audit trail. However, the ISSO must ensure that the chosen pre-selected events are in support of their site's security policy. The TCB should also provide a *post-selection* mechanism which allows the auditor to query/filter/retrieve specific auditable events from the audit trail in a binary form, and output the results of the query in a human-readable format. The advantage of post-selection is that the audit trail can be stored in a binary format that requires less storage space, and queried at any time.

B. OVERVIEW OF AUDITING IN TRUSTED SOLARIS 1.1

As described in Chapter II, the system is configured with a NFS/Audit server and two clients. Both clients *mount* the audit file system from the audit server. This ensures that all audit trail information is stored in one location so that the ISSO can analyze all the audit data concurrently. If each client machine were to store its own audit data, the ISSO would have to analyze each audit trail separately. This would not provide a global view of the system, and illicit activities occurring on separate machines might possibly be overlooked.

All security-relevant actions taken by users, access-control decisions, administrative, and certain other actions are auditable. Each user is assigned an *audit ID* and a *process pre-selection mask* at login for auditing purposes. The audit ID is a unique

number that is assigned by the auditing mechanism when an authorized user begins a session on the system (login) and propagates to all subsequent child processes and across remote logins. The audit ID differs from the user ID because users are allowed to have simultaneous sessions on multiple machines; therefore, the audit mechanism can monitor and compile a separate audit trail for each session. In short, the user ID is for login purposes, and the audit ID is for auditing operations. The process pre-selection mask determines what classes of events will be audited for a process based on the systemwide default audit flags set by the ISSO. The process pre-selection mask consists of two 32-bit binary values that specify what classes of events will be audited when events fail and when events succeed respectively.

An individual audit trail record contains information, generated by the audit mechanism, about one auditable event. Each audit record is put into a buffer in the kernel called the audit queue, and then transferred from the audit queue to the audit trail file by the audit daemon process. The following information is contained in an audit record: attributes describing the objects involved in the audit event, type and time of the event, the user causing the event, and other event dependent data. [Ref. 8]

C. AUDITING TECHNIQUES AVAILABLE TO THE ISSO

In Trusted Solaris the ISSO is responsible for configuring and monitoring auditing, and managing and analyzing the information stored by the audit mechanism. As prescribed in the NCSC Audit Guide [Ref. 14], Trusted Solaris provides the ISSO with *pre-selection* mechanisms for configuring and monitoring auditing, and *post-selection* mechanisms for managing and analyzing the audit information.

The ISSO specifies (pre-selects) the auditable events in the system in two ways. First, the default classes of events are specified in the */etc/security/audit_control* file. An *audit flag* is a two letter designator for the class of auditable events. For example the "fa" audit flag includes all classes of events that involve file attribute accesses, such as the system call commands: *stat*, *statfs*, *truncate*, etc. A few examples of the audit flags are provided in Table 1. The audit flags entered in the *audit_control* file apply to all users'

processes. Second, the ISSO can modify what is audited for individual users by placing audit flags in a user's entry in the */etc/tfm/nis/passwd.adjunct* file on the Network Information Services (NIS) server. A plus (+) must also be entered at the end of the */etc/security/passwd.adjunct* file on every machine, because at login the NIS first checks for entries in the local machine's *passwd.adjunct* file; if no entries are found and the plus (+) is at the end, the NIS then refers to the */etc/tfm/nis/passwd.adjunct* file on the NIS server. The user's individual audit flags are then combined with the system's default audit flags to define the pre-selection set for the user's session. Placing the audit flags in the NIS server *passwd.adjunct* file only, maintains the single-system image. Any audit flag entries found in the local machine's *passwd.adjunct* will be combined with those found in the NIS server's file; however, the NIS server entries override any contradicting entries in the local machine, unless a plus (+) is not found at the end of the file.

The two *post-selection* mechanisms available to the ISSO for managing and analyzing the audit information is *auditreduce* and *praudit*.

> The *auditreduce* program merges audit trail files from each machine to form the systemwide audit trail and selects records according to specified criteria such as time, record type, user, or label. The *praudit* program prints the audit records in a human-readable form. Using *auditreduce* and *praudit* together provides a way to filter and display audit data. [Ref. 8, pp. 72]

Flag	Audit Class	Event Characteristics
all	all	All flags set
fa	file_attr_acc	Access of object attributes: stat, pathconf, etc.
fw	file_write	Write of data, open for writing, etc.
lo	login_logout	Login and logout events
pr	use_of_priv	Use of privilage
sl	set_label	Set file / process security attributes
tf	tfm	Trusted facility management

Table 1: Audit Flags, Class Names, and Descriptions. After Ref. [8]

VI. ANALYSIS

A. AUDITING

1. Volume of Audit Data

As described in the previous chapter, Trusted Solaris provides the ISSO with *pre* and *post-selection* tools of auditable events. However, it is an accepted fact that additional analysis tools are needed, due to the volume and heterogeneous nature of the audit trail data. [Ref. 16][Ref. 17]

The detection of an illicit activity on the system requires extensive and timely analysis of the audit data by the ISSO. Without an additional audit trail analysis tool an illicit activity may easily be overlooked. For example, the *auditreduce* command shown in Figure 3 produces a post-selection listing of just one user, for one audit class (file attribute accesses), over a 20 minute period, piped through *praudit* to format the audit events in a "human" readable form, and finally output to the printer. The printout of the audit data resulting from this command required nearly a ream of paper.

auditreduce -u lowuser -c fa -a 199501101200 -b 199501101220 \| praudit \| lpr

Figure 3. Example of *post-selection* command.

After two users (with sessions at different security levels) completed exploiting the use of the file system modulation covert channel, the post-selection command shown in Figure 3 was performed for the period during which the illicit activity occurred. Although the time and type of illicit activity to look for was known, it still required over half an hour to locate the activity in the output audit data, and this was a listing for only one user over a 20 minute period! Had the post-selection listing been acquired for both users, the listing would have approximately doubled. Though Trusted Solaris does merge the auditable events according to their time of occurrence, the events unrelated to the illicit activity (intermixed with the illicit events) make it difficult for the ISSO to locate and identify the occurrence of the exploitation.

The *pre* and *post-selection* mechanisms provided with the system are insufficient tools to assist the ISSO in managing and analyzing the audit trail information. Therefore, an additional automated auditing tool should be used by the ISSO to assist in the timely and accurate analysis of the audit data to assure all illicit activity is detected. One such tool available for use with the Trusted Solaris system is the Computer Misuse Detection System (CMDS). [Ref. 16]

2. Effect on System Performance

The auditing mechanism should have a minimal effect on the functionality of the system. However, "some performance impact due to auditing is generally inevitable." [Ref. 3] The auditing effects on performance should not be so great that the users encourage administrators to remove or alter the auditing scheme to improve performance. On a good system, the users will notice minimal effects of auditing.

Although the effect of auditing on Trusted Solaris' performance is not substantially noticeable to the user, a timing experiment revealed that there is in fact an effect on performance from auditing. The runtime of the file system modulation covert channel code was timed with only the login and logout (lo) audit class and then with the "all" auditing class on. The portion of code timed did not include any overhead instructions. The "low" user's process was not timed because its runtime is dependent on the "high" user's processing time of the illicit information (256 characters). The results show that there is approximately a 50 percent runtime increase with all auditing turned on (0.63 vs. 0.92 milliseconds). However, rarely will all classes of auditing be pre-selected because of reasons addressed earlier in Chapter V.

B. BANDWIDTH ESTIMATION

The *bandwidth* of a covert channel, usually expressed in bits per second, is the amount of information that can be passed through the illicit channel over a fixed time period. The precise bandwidth measurement of the discovered covert channels is beyond the scope of this thesis. Therefore, a covert channel's bandwidth will either be referred to as being large (high exchange rate of data bits per second), or small (low exchange rate of

data bits per second. The interested reader is referred to *A Guide to Understanding Covert Channel Analysis of Trusted Systems* [Ref. 18].

C. COVERT CHANNELS

1. File System Modulation

This is a large bandwidth covert channel. As discussed in section A.2, the transmission of 256 characters takes less than a second even with all auditing enabled. Additionally, the covert channel is auditable. The two repeated temporal commands to identify the exploitation of this covert channel in the audit trail are *truncate* and *fstatfs* or *statfs*. Auditing of the class "fa" (file attribute accesses) must be enabled to detect these two events. The Trusted Solaris manual [Ref. 8] recommends auditing this class of events.

If the disk resources are available, the channel can be closed if separate */home* file system partitions are created for each level of users on the multilevel system. For example on a military system, unclassified users would have a different */home* directory from secret users, etc. Further partitions could be created for compartments. This would prevent low and high security level users (or a Trojan Horse) from having access to the same file system information; yet, in compliance with the BLP security model [Ref. 6], the high security users could read lower security information, but low security level users can only write (not read) high security information.

2. Memory Space

The use of memory space has been reduced to a *process existence test*, effectively eliminating the covert channel. The bandwidth is limited to the speed by which processes can be created and terminated. The audit class "pc" (process operations) must be enabled and the *recvmsg* event in the audit trail header will identify this operation. There is no way to close this channel unless a system were to contain separate resident memory devices (not merely separate space) for each sensitivity level.

3. Allocation/Deallocation of Devices

This channel has been reduced to a *device existence test*, effectively eliminating the covert channel. The bandwidth is limited to the speed by which devices can be allocated and deallocated. The allocate and deallocate auditable events are found in the audit trail if the audit class "ma" (MAC events) is enabled. There is no known way to close this channel without removing device allocation privileges from users.

VII. CONCLUSION

A. COVERT CHANNELS ON TRUSTED SOLARIS 1.1: INITIAL FINDINGS

This thesis research has provided an initial investigation into the existence of covert channels in the Trusted Solaris 1.1 TCB system. A thorough investigation of all covert channels existing in a system is difficult without detailed top-level specifications or TCB source code. However, this research shows that covert channels exist and can in fact be exploited in the system and the potential of risk exists for systems used in an operating environment with a wider range of TCSEC class systems.

The amount of information generated by the audit mechanism can quickly overwhelm the ISSO and cause detection of illicit activity to be overlooked. The *pre* and *post-selection* mechanisms provided with the system are insufficient tools to assist the ISSO in managing and analyzing the audit trail information. Therefore, an additional automated auditing tool should be used by the ISSO to assist in the timely and accurate analysis of the audit data to assure all illicit activity is detected.

To exploit covert channels in Trusted Solaris a set of tools was carefully built applying software engineering techniques. The disciplined approach to software development ensures the logical *correctness* of software systems. It is especially important for TCB systems to be structured into well-defined largely independent modules; this enables the system to be properly evaluated and provide sufficient assurance that it enforces the security requirements. Although the covert channel system could have been designed without following all the rigors of software engineering, it did result in a highly modularized software system which is easily maintained and logically correct.

B. FUTURE RESEARCH

The following is a list of suggested follow-on work to this thesis:

1. Covert Channel Analysis

Acquire the TCB source code for the Trusted Solaris 1.1 system and conduct further covert channel analysis.

2. Multi-Network System

Two additional *SPARCstation 10* (sun4m) workstations (one with the necessary hardware to be configured as a server) are available that could be configured as a separate distributed system. The two distributed systems could be interconnected and further covert channel analysis conducted in the expanded configuration. Additionally, this would enable yet another researcher to become familiar with the configuration, installation, and administration of a multilevel distributed system.

3. Auditing Implications / Techniques

Conduct further analysis of the auditing implications on system performance and administrative personnel. The design of an audit trail analysis tool would be beneficial to the DoD agencies currently using the Trusted Solaris system.

4. Cryptographic Methods

Explore and analyze the use of different cryptographic methods with covert channels of varying bandwidths.

5. Information Theory

Research and design the synchronization between covert channel processes that would ensure minimal data loss, taking into consideration information theory.

APPENDIX A. UNDERSTANDING TRUSTED SOLARIS 1.1

This appendix contains the specifications for the modules used by the High and Low users for the file system modulation covert channel. All code specific syntax is written in the C language.

Understanding Trusted Solaris 1.1

by Albert Wong
Computer Science Department, Naval Postgraduate School
Nov 8, 1994

Introduction

Trusted Solaris 1.1 is a secure distributed computing system where workstations are interconnected via a local network in a client/server relationship under a single administrative domain using Network Information Service (NIS). Each workstation is based on the SPARC architecture configured with a modified SunOS 4.1.3 operating system and a modified OpenWindows environment. These modifications collectively referred as Trusted Computing Base (TCB) provide the mechanisms to enforce discretionary and mandatory security policies that meets the TCSEC class B1 requirements. In addition, the Trusted Solaris provides a Trusted Facility Management system (TFM) with administrative programs, operating procedures, and window tools to ensure that TCB is safely configured and maintained.

The purpose of this document is to unscramble volumes of Trusted Solaris manuals to provide a basic understanding of how the system works and how to work with the system. The figure attached to the end of this document gives an excellent overview. This diagram is taken directly from the TFM manual. It shows how normal users and users in TFM roles interact with the system. In this document, we will discuss the diagram in some detail. Along the way, a number of security concepts and features will emerge. For more information on a particular topic, refer to the following Trusted Solaris manuals:

- Security Features Users' Guide,
- Trusted Facility Management Manual Volumes I and II,
- OpenWindows User Training Tutorial Guide.

Normal Users

Normal users are users who have no special authorization that would allow them to bypass the security mechanisms of the system. They can run any program as along as the program requires no special privilege and has no security implication. Special authorizations are granted to users to run a privileged program and to assume an administrative role. As shown in the diagram, a normal

user can run untrusted programs, trusted SunOS programs, and forced privilege programs. Untrusted programs are programs that require no privilege. Typically, these programs are programs that a normal user may develop or install but oddly enough, they encompass most of SunOS system commands including the shell. Trusted SunOS programs are those system commands that require authorizations for users to run them and privileges to do the work when they are running. Forced privileges programs are equivalent to setuid 0 programs; the required privileges are coerced during the execution of the program. Nevertheless, programs invoked by normal users must access the kernel modules via the *unrestricted uses of system calls*. The *unrestricted uses of system calls* is a separate set of system calls whereby access control in the kernel is strictly enforced so that security of the system cannot be compromised.

Authorized Users in Roles

In a traditional UNIX system including SunOS, the administration of the system belongs to a root user often referred to as a superuser. In Trusted Solaris, there is no superuser. The tasks and power of the superuser are distributed among different roles. TFM provides a set of guideline for the administration of the system. TFM is based on two fundamental principles.

- **Separation of Duties**
  System administration tasks are divided among roles to reduce human errors and wrongdoings,

- **Principle of least privilege**
  Each user or process is granted the most restricted set of privileges that a user or a process needs for the performance of the authorized task.

TFM roles are assumed by authorized users to perform specific system functions. The tasks performed by each role are restricted to a portion of the total administrative tasks. See Authorizations and Privileges below. The duties and responsibilities of the TFM roles are:

- **Information System Security Officer (ISSO)**
  The ISSO enforces on-site security policy and performs security relevant tasks such as setting up labels, managing user security, administering audit events, assigning privileges to programs and assessing new applications.

- **System Administrator**
  The system administrator is responsible for the standard system administrative tasks such as setting up user accounts, managing machines and network, maintaining files and file systems, administering window tools, and installing new software.

- **System Operator**
  The system operator is responsible for managing devices such as tapes and printers, starting and shutting down the system, and backing up files.

As shown in the diagram, authorized users in roles can run trusted SunOS programs, forced privilege programs, and administrative programs provided by TFM but not untrusted programs. Programs invoked by authorized users in roles access the kernel modules via the *restricted uses of*

system calls which allows them to bypass the security mechanisms.

To assume a TFM role, the administrator must first login as normal user. The user, then, selects the *Trusted Path* menu to assume the appropriate role. You cannot login as a TFM role and you cannot assume a TFM role unless you are authorized by the ISSO for that role.

Login Process

After the system is rebooted, system automatically comes up. The ISSO must first login as normal user and assume an ISSO role to enable logins for all other users. To login to the Trusted Solaris system, you need an account on the system. The system administrator creates the account but the ISSO sets up the initial password, and the security attributes for your account.

The login process consists of a series of three trusted login window screens. The first login screen allows you to enter your login name and password; the second screen is read-only that provides general information about your workstation; and the third screen allows you to establish your session clearance. After login, you are automatically operating in Trusted Solaris OpenWindows environment and allowed to perform operations within your session level.

Password

Password is used for authentication. It authorizes the user to access the system. It is the first line of defense against system penetration by unauthorized users. Trusted Solaris provides the required protections to ensure that your password cannot be traced when you enter the system at login or reenter the system through lock screen. Other mechanisms such as system generated password, shadowed password, and password aging are supported.

Working in the Trusted Solaris Environment

After login, the workspace on the screen consists of two special items: the console tool and the screenstripe. Other optional items such as file manager, wastebasket, clock, mail tool and printing tool may be included.

- **Console Tool**

Console tool is a read-only text window used for displaying error and system messages.

- **Screenscripe**

The screenstripe is a narrow rectangular stripe located across the bottom of the screen. It is a permanent part of the root window that cannot be suppressed. The screenstripe appears after login and remains throughout the session monitoring the status of the input devices and the trusted programs.

==========

If the screenstripe appears on the trusted login screen and the trusted lock screen, your are being spoofed -- Someone is trying to capture your password.

==========

Screenstripe Components

The screenscripe consists of two bands. The upper band displays information about the key-

board. It consists of a keyboard grab symbol, a trusted shield, and the input information label that identifies the security level of the input from the keyboard. Likewise, the lower band displays the information about the mouse. It consists of a mouse grab symbol, a trusted shield, and the mouse label of the window that is holding the mouse pointer. The grab symbol shows the grab status. A grab occurs when an application take control of the input. Keyboard grabs can mean tampering and have security implications The trusted shield, when appears, indicates that the current application is trusted. The screenstripe has, in addition, a hidden component called the trusted path.

Trusted Path

Trusted path is a mechanism that allows users to perform security related operations in a safe manner. It provides users with an unforgeable connection to the TCB and prevent users from being spoofed by trojan horses and other misleading programs that tamper with the security of the system. A trusted path can be activated in one of two ways:

- **Using the Screenstripe**

By pressing the menu button which is the right mouse bottom on the screenstripe, a trusted path menu will appear. On this menu, you can select such items as:

  - ***Utilities***

  to refresh the screen, to control window size and display, to save current workspace layout, to lock the screen, to logout, and to shut down the machine (but you won't be able to bring it back up unless you are authorized).

  - ***Set Labels***

  to set labels on files and directories, to change the label on the workspace in order to bring up window tools with a proper label, and to cut and paste between windows (see Transfer Data in OpenWindows).

  - ***Change Password***

  to change your password.

  - ***Set Screen Access***

  to modify your monitor access control list to enable other users to access your screen.

- **Trusted Frame Menu**

By pressing the menu button on the dark stripe above the header of any window or icon, a trusted frame menu appears. On this menu, select

  - ***Show Full Screen Label***

  to open a read-only window to show the extended label for that window or icon, or select

  - ***Set Input Information Label*** (IIL)

  to set information label on input from your keyboard. The information level you choose is, of course, constrained by your session clearance and your accreditation range.

- **Workspace Menu**

From the workspace menu, you can run deskset tools, allocate and deallocate devices, and change screen properties. To bring up the workspace menu, press the menu button anywhere on the background areas of the screen. If you select the ***programs*** item on the workspace menu, you have a complete list of deskset tools including the shell.

Session Clearance

Session clearance is the highest sensitivity level at which you can work during a particular login session. The ISSO sets the default clearance when defining the security attributes of your account. You can choose to lower your session clearance when you login. See **Login Process** above.

Accreditation Range

Accreditation range is a set of authorized sensitivity levels for which access can be granted. There are two types of accreditation ranges: system and user:

- **System Accreditation Range**

System accreditation range is set of valid sensitivity levels on the system defined by the limits of SYSTEM_HIGH and SYSTEM_LOW. SYSTEM_HIGH is above all other sensitivity levels in the system. This label is used to protect system files that contain security sensitive information since normal users cannot read up. SYSTEM_LOW is lowest level dominated by all other sensitivity levels in the system. This is used to protect publicly accessible system files and system commands since normal users cannot write down (see Mandatory Access Control below for enforcement of security policy).

- **User Accreditation Range**

User accreditation range is a subset of the system accreditation range excluding both SYSTEM_HIGH and SYSTEM_LOW. By default, your session clearance and your home directory mark the highest and the lowest sensitivity levels at which you can work. When you login, the default security level or label (SL) is set at the level of your home directory which is minimum. Your default SL is the SL of the workspace menu. This SL will, therefore, be inherited by all processes invoked from the menu and hence all the files created by those processes. To ensure the proper labels on these files, you can change your default SL to the desired level before invoking the process providing the desired level is within your accreditation range and does not exceed your current session clearance.

To change your default SL, you change the SL of your workspace using the *Set Labels* option from the *Trusted Path* menu. Recall that the mouse label at the screenstripe displays the label that holds the mouse. Using this feature, you can display your default SL by moving your mouse on the workspace which is the background of your screen. Likewise you can display the label of your session clearance by moving the mouse pointer to the screenstrip.

Labels

Labels represent the levels of sensitivity associated with the system entities. Trusted Solaris

assigns labels to every subjects and objects to regulate the flow of information. Each label has two components displayed as IL[SL] in the label stripe above the title bar of the window.

IL is the information label that describes the security level of the information contained within the entity. Its purpose is track the flow of information, not to enforce access control. IL comprises a classification, a set of compartments as well as marking and caveats for the handling of entity.

SL is the sensitivity label consisting of a classification and a set of compartments that are used as the basis for mandatory access control decisions.

- **Classifications**

  Classifications represent hierarchical levels of security such as TOP SECRET, SECRET CONFIDENTIAL, AND UNCLASSIFIED. TOP SECRET is the highest level of classification which dominates all other classifications. Each label has one classification out of possible 16 classifications that Trusted Solaris supports.

- **Compartments**

  Compartments are nonhierarchical and independent of classifications. Each compartment has a name representing a work group, a project or a topic. A label has one classification and a set of zero or more compartments. One label dominates another label if the other label is a proper subset. If neither label dominates the other, the two labels are disjoint and noncomparable. Trusted Solaris supports a maximum of 128 separate compartments.

- **Markings**

  Markings are used to regulate the flow of information. They describe how data should be handled rather than how it should be protected. Markings are the third component of an IL and are not included in SL. An IL has zero or more markings. Examples of a marking are NOFORN for *No Foreign Dissemination* and LIMDIS for Limited Distribution to members of a particular project.

Access Control

Access control is accomplished by both discretionary and mandatory access controls. They reflect two distinct aspects of the overall security policy. Both policies must be satisfied in order to gain access to the object.

- **Discretionary Access Control (DAC)**

  DAC is based on user identity and need-to-know criteria. Its implementation is the same as in standard UNIX. DAC also includes an access control list (ACL) to allow file owner to restrict access permission by individual users or groups. ACL is a list entries consisting of two parts as depicted as follows:

Part 1

#file:	filename	name of the object
#owner:	ownername	name of the owner
#group:	groupname	name of owner's group

user::rwx	owner's permission
user:name:r-x	individual user's permission
. . .	
group::r-x	owner's group permission
group:groupname:-w-	individual group's permission
. . .	
class:r-x	permission mask for users and groups
other:--x	permission for others

Part 2

default:user::rwx	Part 2 is applicable for directories only. These default ACL
. . .	settings are used for files created under this directory.

ACLtool which is a window-based management tool can be used to construct the entries. It can be invoked from the workspace menu. DAC access control is based on ACL. If an object does not have an ACL, standard UNIX permission bits are used.

- **Mandatory Access Control (MAC)**

MAC, on the other hand, provides access control based on the sensitivity labels between two objects. MAC is checked before DAC. Because users cannot modify sensitivity labels to affect the Mac check. MAC, therefore, provides stronger level of protection.

MAC enforces a *write up/read down* (WURD) policy. Read access requires that the subject SL dominates the object SL; write access requires that the subject SL be dominated by the object SL; and read/write or modify access requires that the SLs be equal between the subject and the object.

Authorizations and Privileges

Authorizations are given to users and privileges are assigned to programs. In order for a user to assume a TFM role or to perform a trusted function, that user must have the proper authorization. By the same token, a TCB program to perform must have the proper privilege to perform the trusted function on behalf of the user. These trusted functions can have profound security implications and must be performed by trusted programs and trusted users. The ISSO uses trusted window tools to assigned authorizations to the designated users according to the security policy. Authorizations are checked by the privileged programs before performing the trusted functions. Authorizations and privileges in Trusted Solaris are the means to distribute the system responsibilities so that each user and each program is granted the proper authority needed for the performance of the assigned task. There are two types of authorizations:

TFM Role Authorizations

- TFM _ROLE_isso for the ISSO role
- TFM_ROLE_admin for the system administrator role
- TFM_ROLE_oper for the system operator role
- TFM _ROLE_root for the ISSO to install application software

SunCMW Authorizations

- SunCMW_boot_system to allow users to enable logins after reboot.
- SunCMW_terminal_login to allow users to login to a terminal attached to the system to perform maintenance.
- SunCMW_remote_login to allow users to telnet, ftp, and rlogin to remote hosts.
- SunCMW_upgrade_sensitivity_label to allow users to upgrade the SL of an object for moving data between windows and relabeling files.
- SunCMW_downgrade_sensitivity_label to allow users to downgrade the SL of an object for moving data between windows and relabeling files.
- SunCMW_outside_accreditation_range to allow users to operate outside the user accreditation range.
- SunCMW_set_single_level_device to allow users to allocate and deallocate devices to import or export data.

Privileges are grouped into an array called privilege set. Each element of the privilege set corresponds to a single privilege that allows or disallows the program to bypass certain security restriction (i.e., file_mac_read). Out of a total of 128 possible privileges, 78 are currently defined. There two privilege sets associated with each program:

- allowed
  Allowed privilege set defines the individual privileges that a program can inherit from its parent process.

- forced
  Forced privilege set defines the individual privileges that a program has independent from its parent process.

Managing Files and Directories

In Trusted Solaris, managing files and directories is not just a matter of organizing; it is also a matter of classifying the information you are handling. We conclude this document by discussing some of the basic operations but first we summarize the issues regarding security rules and file attributes.

- **Security Rules**

When you try to access a file or directory, two common error messages are often occurred:

- Not Owner -- indicates that the object SL is above your session clearance, and
- Permission Denied -- indicates that your session clearance dominates the SL of the object but not the process trying to access the object.

The following security rules are reiterated:

To create a file or directory within a directory, it requires

- MAC read and DAC search access to all the directories in the path,
- MAC write and DAV write access to the directory in which the entry is to be created.

To open a file in a directory, it requires

- MAC read and DAC search access to all the directories in the path.

To list the files a directory, it requires

- MAC read and DAC search access to all the directories in the path,
- DAC read access to the directory to be listed.

MAC rules with respect to labels:

- In any object, the IL must always be dominated by the SL.
- SL of an object is inherited.
- IL of an object can be floated.
- All labels used in the system must be defined in the system accreditation range.
- All labels used by a normal user must be contained in the user accreditation range.
- SL of a home directory is crated at the lowest level.
- SL of a subdirectory dominates the SL of its parent directory.
- SL of a program that creates a file must be equal to SL of its directory.

- **File Attributes**

There are three types of file: ordinary, directory, and special. Ordinary files are files such as data, source and object code, and executable; directory files are files that contain other file and subdirectories; and special files are devices such disks, tapes, and printers. Each file has its own set of attributes. For example:

- DAC including ACL
- MAC labels including both IL and SL
- privileges if the file is executable.

- **Creating a file or directory**

Just as in standard UNIX, there are many ways to create a file or directory. A file can be created from an editor or from the file manager. A directory can be created from a command line using a command tool or from the file manager. The file you created inherits both the IL and the SL from the tool you use (remember directory has no IL). To ensure the file you created has the desired label, you can choose the tool that has the right label or change the label after creation using the *Set Label* option from the *Trusted Path* menu

DAC components such as permission, owner, group, and ACL require very little attention because the default values are in general acceptable. When a file or a directory is created, DAC components are derived from ACL. Recall that ACT has two parts. Part 2 is used as default ACL for files created under that directory. If a directory is created, part 2 from the parent directory is appended so that the default can be propagated to its subdirectories. In the absence of ACL, standard UNIX implementation is used where permissions are derived from the creation mask called *umask*.

- **Copying or Editing files**

When your are copying or editing a file, the DAC attributes remain unchanged but the MAC attributes of the file are likely to change. If a file is copied to another file in the same directory, the file attributes are preserved.

If you are editing a file, the SL of the edited file is inherited from the SL of the editor and the IL will be floated. **Floating IL** is a phenomenon that occurs when an operation is performed between two objects with different ILs. The merging of the two ILs is resulted in conjunction where conjunction is the maximum of the classification portion of the two ILs and the union of their compartments and marking.

For example, If an editor has a null IL editing a file with an IL of CONFIDENTIAL and NOFORN. The IL of the editor will be floated up to the level of the file and the edited file inherits the IL of the editor. The file, at this point, has the same IL as before. Now if the same editor is used to edit another file with an IL of UNCLASSIFIED. The IL of the second file will be changed from UNCLASSIFIED to CONFIDENTIAL and NOFORN. To retain the original IL for the second file, a fresh copy of an editor must be used.

- **Multilevel Directories**

It is possible to create directories that let you create files with different of SLs. One way to accomplish this is the use of Multi-label directory (MLD). Under the MLD, you can create files with different classification levels. The way Trusted Solaris works is to collect the like-labeled files into a hidden single-label directory (SLD) under a given MLD. If you list the contents of the directory, the only files you see are the files with SLs equal to the SL of the tool you used. To create a MLD, use the command *mkdir -M directoey-name*. If you cd to that directory, you can create files with different SLs by changing your default SL.

A more direct approach is to create the directory and relabel it. This approach does, however require authorization for upgrading the SL label. It is because the SL of the parent directory is lower than the SL of the label that you try to change. You will not be able to save the label in the parent directory unless you have authorization.

- **Transferring Data in OpenWindows**

Transferring data in an OpenWindows environment amounts to moving text from one window to another. These operations include *cut and paste* and *drag and drop*. With the required authorization, you can copy and paste up or down to windows with different SLs.

- **Cut and Paste**

Cut and paste operation in a window can occur regardless of labels. All users can cut and paste data between windows with equal SL.

- **Drag and Drop**

Drag and Drop operation requires ether the SLs of the objects be equal or the SL of the destination window dominates the SL of the source window. You can drag a file to the open part of a window and drop the file into the window. You can also drag and drop a file to an icon. When you drop a file to an icon, it replace what was there.

- **Upgrading and Downgrading Text**

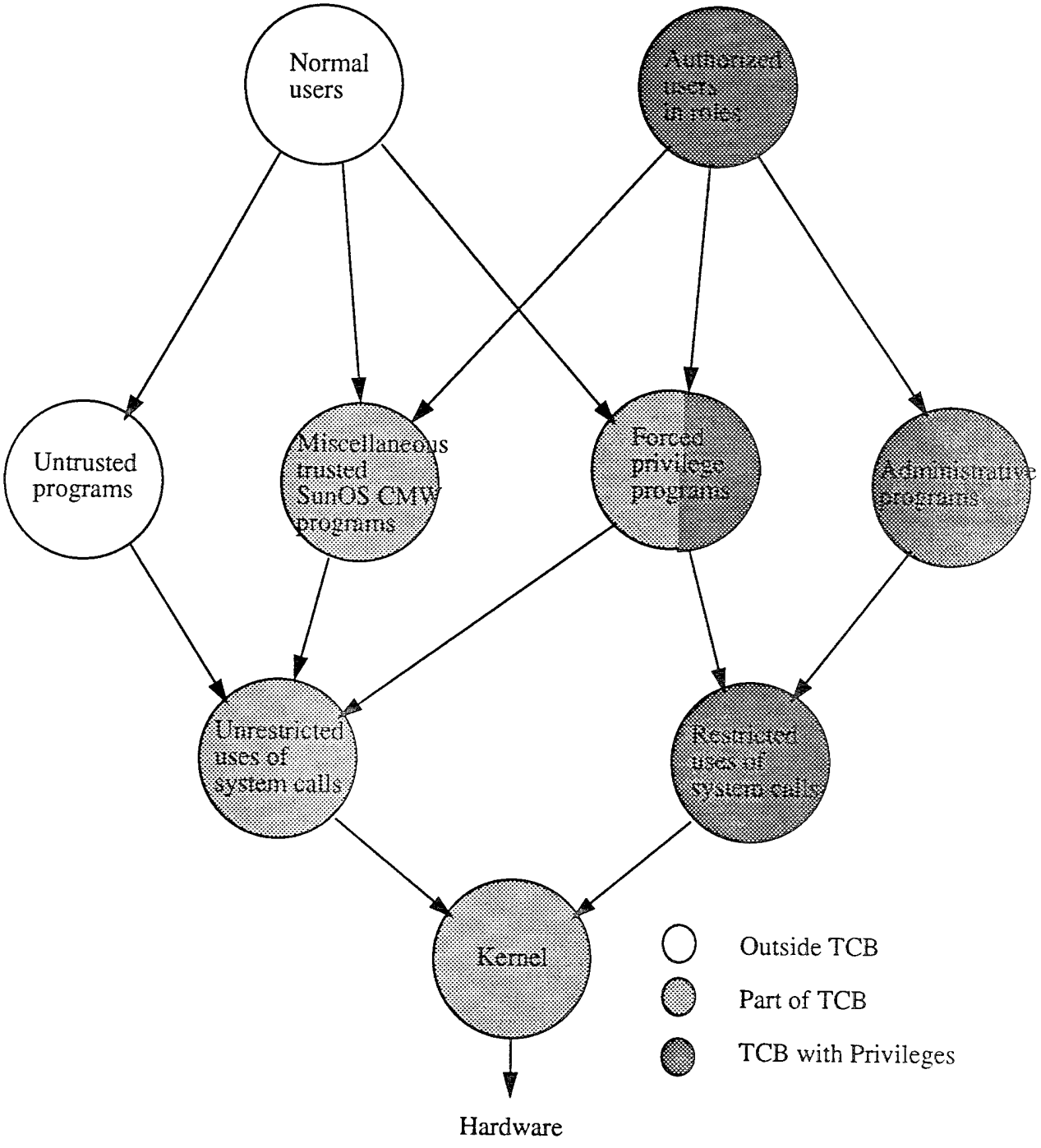
One reason to upgrade and downgrade text is to produce a report for people with different clearances. Before you can perform a upgrade or downgrade, authorization to upgrade or downgrade is required. When upgrading text, the IL of the destination label merges and

floats up. You may need to create a new IL to reflect the actual IL of the upgraded data. When downgrading text, you need to create new IL for the text you are transferring. if the destination SL is lower than the source IL.

To upgrade or downgrade text, use the copy and paste operation as follows:

1 - bring up two editors with two different label,
2 - edit a file using one editor depending on whether you are updgrading or downgrading text,
3.- highlight the text you wish to copy,
4 - Paste the text into the other editing window. A *Selection Labeler* window pops up and indicates that you are upgrading or downgrading information.
5 - Select an appropriate IL with the popped up Selection labeler.

Users Interactions with TCB



APPENDIX B. MODULE SPECIFICATIONS

This appendix contains the specifications for the modules used by the High and Low users for the file system modulation covert channel. All code specific syntax is written in the C language.

A. COMMON CONFIGURATION SYSTEM PARAMETERS

A.1 MODULE CONSTANTS:

```
char QUIT = '9'
int NOERROR = 0
int EXCEPTION = -1
int TRUE = 1
int FALSE = 0
int MAX_MENU_LENGTH = 80 -- max length of menu lines
int MAX_PATH_LENGTH = 256 -- max length of path names
int MAX_SEGMENT_SIZE = 256 -- processing array size
int ERROR_LINES = 2 -- number of lines in error prompt
int EOF_SIGNAL = -1 -- negative value to signal EOT
```

TYPES:

```
MENU_LINE_type -- character array of length
    MAX_MENU_LENGTH
    typedef char MENU_LINE_type[MAX_MENU_LENGTH]
PATHNAME_type -- character array of length
    MAX_PATH_LENGTH
    typedef  char PATHNAME_type[MAX_PATH_LENGTH];
SEGMENT_type -- character array of length
    MAX_SEGMENT_SIZE
    typedef char SEGMENT_type[MAX_SEGMENT_SIZE];
CODE_type -- integer array of length MAX_SEGMENT_SIZE
    typedef int CODE_type[MAX_SEGMENT_SIZE];
```

VARIABLES:

```
MENU_LINE_type  menu_error[ERROR_LINES] = {
    "ERROR: Invalid menu choice.",
    "Press <ENTER> to continue."
};

MENU_LINE_type  path_error[ERROR_LINES] = {
    "ERROR: Invalid pathname.",
    "Press <ENTER> to continue."   };
```

A.2 DATABASES:
None.

B. HIGH DISPLAY MODULE

B.1 MODULE CONSTANTS:
None.

TYPES:
None.

VARIABLES:
None.

B.2 DATABASES:
None.

B.3 Display displays all the user menus and prompts for the functions to the monitor.

B.3.1 INPUTS
Menu: storage structure for passing menu or prompt character strings.
Lines: number of lines in Menu to display.

B.3.2 PROCESSING
Initialize Status to NOERROR
Loop incrementing through Menu for number of Lines while Status equals NOERROR
  Call UNIX C-library *printf* with parameter character string from Menu and set Status equal to returned function value.
return Status

B.3.3 EFFECTS
None.

B.3.4 EXCEPTIONS
UNIX exceptions.

B.3.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

B.3.6 DATABASE REFERENCED
None.

B.3.7 INTERFACE: int Display(
MENU_LINE_type *Menu,
int &Lines);

C. HIGH CONFIGURATION SYSTEM PARAMETERS

C.1 MODULE CONSTANTS:

int PATHS = 5 -- number of files used for truncating
int MAX_BLOCK_SIZE = 1024 -- block size of file reads

TYPES:

BLOCK_type -- character array of length MAX_BLOCK_SIZE
typedef char BLOCK_type[MAX_BLOCK_SIZE];

VARIABLES:

None.

D. HIGH MAIN MODULE

D.1 MODULE CONSTANTS:

int MAIN_MENU_LINES = 4
char PROCESS = '1'

TYPES:

None.

VARIABLES:

```
MENU_LINE_type main_menu[MAIN_MENU_LINES] = {
    "MAIN MENU",
    "1. Process a file.",
    "9. Quit, exit to system prompt.",
    "Enter your choice: "
};
```

D.2 DATABASES:

None.

D.3 High_main loops calling **HCC_Covert_init** and **HCC_Covert_run** until the user is done processing files.

D.3.1 INPUTS

None.

D.3.2 PROCESSING

Local Variables:

char Choice = PROCESS; -- variable to store user's main_menu choice

char Covert_choice; -- variable to store user's covert_menu choice

Initialize Status to NOERROR

Loop while Choice is not equal to QUIT

Call Display with parameters main_menu and MAIN_MENU_LINES and set Status equal to returned function value.

If Status equals NOERROR

Call UNIX C-library *getchar*, set Choice equal to returned character and set Status equal to returned function value.

If Status equals NOERROR and Choice equals PROCESS

Call HMC_Covert_init, set Covert_choice equal to returned character and set Status equal to returned function value.

If Status equals NOERROR and Covert_choice equals QUIT

Set Choice equal to QUIT

If Status equals NOERROR and Choice equals PROCESS

Call HMC_Covert_run with no parameters and set Status equal to returned function value.

If Status equals NOERROR and value of Choice is invalid

Call Display with parameters menu_error and ERROR_LINES and set Status equal to returned function value.

return Status

D.3.3 EFFECTS

None.

D.3.4 EXCEPTIONS

UNIX exceptions.

D.3.5 OUTPUTS

Status: NOERROR or EXCEPTION of the function.

D.3.6 DATABASE REFERENCED

None.

D.3.7 INTERFACE: int high_main();

E. HIGH_MODULATOR_CONTROLLER MODULE

E.1 MODULE CONSTANTS:

int COVERT_MENU_LINES = 4

int THEORY_MENU_LINES = 4

char PROBABILITY = '1' -- choice for Info_Theory menu

char MODULATE = '1' -- choice for Covert method menu
PROB_DIST -- Probability distribution codes structure
Alpha_codes -- integer array containing the probability distribution codes for the lowercase alphabetic characters.
Num_codes -- integer array containing the probability distribution codes for the numeric characters 0 thru 9.

```
struct Prob_dist{
    int     alpha_codes[26] = {
            2,19,11,10,0,14,16,8,4,23,21,9,13,
            5,3,15,24,7,6,1,12,20,17,22,18,25};
    int     num_codes[10] = {
            26,27,28,29,30,31,32,33,34,35};
};
```

TYPES:

HMC_CDB_type -- HMC_Covert Database structure
Covert_method - integer value indicating covert method to use.
Info_theory_method - integer value indicating info theory method to use.

```
typedef  int HMC_CDB_type[2];
```

VARIABLES:

```
MENU_LINE_type covert_menu[COVERT_MENU_LINES] =
{
    "COVERT METHOD MENU",
    "1. Modulate the file system.",
    "9. Quit, return to Main menu.",
    "Enter your choice: "
};
MENU_LINE_type theory_menu[THEORY_MENU_LINES] =
{
    "INFORMATION THEORY METHOD MENU",
    "1. Probability distribution of characters method.",
    "9. Quit, return to Covert Method menu.",
    "Enter your choice: "
};
HMC_CDB_type  CDB; -- Covert database
```

E.2 DATABASES:

HMC_Covert Database.

E.3 HMC_Covert_init sets the Covert_method in the HMC_Covert database equal to the user's choice. Calls other module initialization functions based on the covert method chosen.

E.3.1 INPUTS

None.

E.3.2 PROCESSING

Local Variables:
    char Theory_choice; -- theory menu choice.
Initialize Status to NOERROR
Set Covert_choice equal to MODULATE
Loop while Covert_choice is not equal to QUIT
    Call Display with parameters Covert_menu and COVERT_MENU_LINES
      and set Status equal to returned function value.
    If Status equals NOERROR
        Call UNIX C-library *getchar*, set Covert_choice equal to returned
          character and set Status equal to returned function value.
    If Status equals NOERROR and Covert_choice equals MODULATE
        Set Covert_method in HMC_Covert database equal to MODULATE
        Call HMC_Info_theory_init, set Theory_choice equal to returned
          character and set Status equal to returned function value.
    If Status equals NOERROR and Theory_choice equals QUIT
        Set Covert_choice equal to QUIT
    If Status equals NOERROR and Covert_choice equals MODULATE
        Call HFR_Init with no parameters and set Status equal to returned
          function value.
    If Status equals NOERROR and Covert_choice equals MODULATE
        Call HMT_Init with no parameters and set Status equal to returned
          function value.
        Set Covert_choice equal to QUIT
  If Status equals NOERROR and value of Covert_choice is invalid
      Call Display with parameters menu_error and ERROR_LINES
return Status;

E.3.3 EFFECTS

HMC_Covert database is initialized to contain covert method to use.

E.3.4 EXCEPTIONS

UNIX exceptions.

E.3.5 OUTPUTS

Covert_choice: character value containing user's Covert_menu choice.
Status: NOERROR or EXCEPTION of the function.

E.3.6 DATABASE REFERENCED
HMC_Covert database.

E.3.7 INTERFACE: int hmc_covert_init(char Covert_choice);

E.4 HMC_Info_theory_init sets the Info_theory_method in the HMC_Covert database equal to the user's choice. Initializes needed structures based on the information theory method chosen.

E.4.1 INPUTS
None.

E.4.2 PROCESSING
Initialize Status to NOERROR
Set Theory_choice equal to MODULATE
Loop while Theory_choice is not equal to QUIT
    Call Display with parameters Theory_menu and THEORY_MENU_LINES
      and set Status equal to returned function value.
    If Status equals NOERROR
        Call UNIX C-library *getchar*, set Theory_choice equal to returned
          character and set Status equal to returned function value.
    If Status equals NOERROR and Theory_choice equals PROBABILITY
        Set Theory_method in HMC_Covert database equal to PROBABILITY
    If Status equals NOERROR and value of Theory_choice is invalid
      Call Display with parameters menu_error and ERROR_LINES
return Status;

E.4.3 EFFECTS
HMC_Covert database is initialized to contain value indicating info theory method to use.

E.4.4 EXCEPTIONS
UNIX exceptions.

E.4.5 OUTPUTS
Theory_choice: character value containing user's Theory_menu choice.
Status: NOERROR or EXCEPTION of the function.

E.4.6 DATABASE REFERENCED
HMC_Covert database.

E.4.7 INTERFACE: int hmc_info_theory_init(char Theory_choice);

E.5 HMC_Covert_run calls the appropriate covert channel method based on the value of Covert_method in the HMC_Covert database.

E.5.1 INPUTS
None.

E.5.2 PROCESSING
Initialize Status equal to NOERROR
If Status equals NOERROR and Covert_method in HMC_Covert database equals MODULATE
  Call HMC_Modulator_controller with no parameters and set Status equal to returned function value.
return Status;

E.5.3 EFFECTS
None.

E.5.4 EXCEPTIONS
UNIX exceptions.

E.5.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

E.5.6 DATABASE REFERENCED
HMC_Covert database.

E.5.7 INTERFACE: int hmc_covert_run();

E.6 HMC_Modulator_controller calls **HFR_Read** repeatedly each time reading the number_of_bytes from the file and then calling **HMC_Info_theory** and **LMT_Truncate**, until the entire file has been read and processed. It then calls **HFR_Close** to close the read file that was opened, and **LMT_Delete** to delete the files used for truncating.

E.6.1 INPUTS
None.

E.6.2 PROCESSING
Local Variables:
  int Bytes_read = MAX_SEGMENT_SIZE;
  int Bytes_to_write;
  SEGMENT_type Read_array;

CODE_type Code_array;
Initialize Status to NOERROR
If Covert_method in HMC_Covert database equals MODULATE
Loop while Status equals NOERROR and Bytes_read equals
MAX_SEGMENT_SIZE
Call HFR_Read, set Read_array equal to the returned array of characters, Bytes_read equal to the returned number of bytes read, and set Status equal to returned function value.
If Status equals NOERROR
Call HMC_Info_theory with parameters Read_array and Bytes_read; set Code_array equal to returned integer array and set Status equal to returned function value.
While Status equals NOERROR increment through Code_array for number of Bytes_to_write
Call HMT_Truncate with value at current Code_array position and set Status equal to returned function value.
return Status

E.6.3 EFFECTS
None.

E.6.4 EXCEPTIONS
UNIX exceptions.

E.6.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

E.6.6 DATABASE REFERENCED
None.

E.6.7 INTERFACE: int hcc_modulator_controller( );

E.7 HMC_Info_theory takes all the significant characters in the Read_array and stores them in the Write_array. Then **HMC_Char_prob_dist** is called to convert the significant characters in the Write_array to the appropriate numeric codes.

E.7.1 INPUTS
Read_array: storage structure for passing the bytes read from the file.
Bytes_read: number of bytes contained in the read_array (this is always positive).

E.7.2 PROCESSING
Local Variables:

```
    SEGMENT_type  Write_array;
Initialize Status to NOERROR
Initialize Bytes_to_write to zero
If Info_theory_method in HMC_Covert database equals PROBABILITY
    Loop incrementing through Read_array for number of Bytes_read
        If character is significant (a-z, 0-9)
            Copy character from Read_array to Write_array
        If character is significant (A-Z)
            Convert to lowercase and store in Write_array
        Increment Bytes_to_write by one
    Call HMC_Char_prob_dist with parameters Write_array and
      Bytes_to_write; set Code_array equal to returned integer array and Status
      equal to returned function value.
return Status
```

E.7.3 EFFECTS

E.7.4 EXCEPTIONS

UNIX exceptions.

E.7.5 OUTPUTS

Code_array: storage structure for passing the converted character codes.

Bytes_to_write: number of bytes contained in the Code_array (this is always positive).

Status: NOERROR or EXCEPTION of the function.

E.7.6 DATABASE REFERENCED

HMC_Covert database.

E.7.7 INTERFACE:

```
int hmc_Info_theory(SEGMENT_type Read_array,
                    int &Bytes_read,
                    CODE_type Code_array,
                    int &Bytes_to_write);
```

E.8 HMC_Char_prob_dist converts the significant characters in the Write_array to the proper integer codes and stores the codes in the Code_array.

E.8.1 INPUTS

Write_array: storage structure for passing the significant characters.

Bytes_to_write: number of bytes contained in the write_array (this is always positive).

E.8.2 PROCESSING

Initialize Status to NOERROR
If Status equals NOERROR
    Loop incrementing through Write_array for number of Bytes_to_write
        If character is (a-z) then convert using Alpha_codes
        If character is (0-9) then convert using Num_codes
        Store code in Code_array
return Status

E.8.3 EFFECTS

None.

E.8.4 EXCEPTIONS

None.

E.8.5 OUTPUTS

Code_array: storage structure for passing the converted character codes.
Status: NOERROR or EXCEPTION of the function.

E.8.6 DATABASE REFERENCED

None.

E.8.7 INTERFACE:

```
int HMC_Char_prob_dist(
                        SEGMENT_type Write_array,
                        CODE_type Code_array,
                        int &Bytes_to_write);
```

F. HIGH FILE READER MODULE

F.1 MODULE CONSTANTS:

```
int READ_PROMPT_LINES = 1
```

TYPES:

```
HFR_OFS_type -- HFR_Open_file Database structure
    typedef  struct {
        int         open_status - boolean value indicating
                      open status of file;
        fsid_t      file_descriptor - integer assigned by the
                      system when a file is opened;
        long        file_size - size of the file in bytes;
        int         more - integer boolean that indicates if
                      more bytes to read;
```

```
        long            tot_bytes_read - total bytes read from
                          the file;
        BLOCK_type      block_array - holds a block of bytes read
                          from the file;
        long            block_bytes_read - number of bytes
                          read into the block_array;
        long            block_index - index into block_array,
                          indicates next byte to read;
    }HFR_OFS_type;
```

VARIABLES:

```
MENU_LINE_type read_prompt = {
    "Enter the path to the file you want to read: "
};
HFR_OFS_type HFR_OFS;
HFR_OFS_type *HFR_OFS_ptr = HFR_OFS;
PATHNAME_type HFR_Pathname;
```

F.2 DATABASES:

HFR_Open_file Database.

F.3 HFR_Init initializes the HFR_OFS Database structure. Gets the pathname of the file to read from the user and calls **HFR_Open** to open the file.

F.3.1 INPUTS

None.

F.3.2 PROCESSING

Local Variables:

```
    int Done = FALSE; -- loop control variable
```

Initialize Status to NOERROR

Loop while Done equals FALSE

- Call Display with parameters read_prompt and READ_PROMPT_LINES; set Status equal to returned function value.
- If Status equals NOERROR
  - Call UNIX C-library *gets*, set HFR_Pathname equal to returned character array and set Status equal to returned function value.
- If Status equals NOERROR
  - Initialize HFR_Open_file database
  - Call HFR_Open with no parameters and set Status equal to returned function value.
- If Status equals NOERROR
  - Set Done equal to TRUE

If Status equals EXCEPTION and UNIX *errno* equals invalid path exception
Call Display with parameters path_error and ERROR_LINES; set Status equal to returned function value.
return Status

F.3.3 EFFECTS
HFR_Open_file database is initialized:
open_status is set to FALSE.
more is set to TRUE.
tot_bytes_read is set to zero.
block_bytes_read is set to zero.
block_index is set to start of Block_array.

F.3.4 EXCEPTIONS
UNIX exceptions.

F.3.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

F.3.6 DATABASE REFERENCED
HFR_Open_file database

F.3.7 INTERFACE: int HFR_init();

F.4 HFR_Open opens a file for reading and sets the appropriate values in the HFR_Open_file database structure.

F.4.1 INPUTS
None.

F.4.2 PROCESSING
Local Variables:
struct stat stat_buffer -- structure for holding return values from UNIX *fstat*
stat *stat_buf_ptr = stat_buffer;
Initialize Status to NOERROR
Call UNIX *open* with parameter HFR_Pathname and set Status equal to returned function value.
If Status equals NOERROR
Set file_descriptor in HFR_OFS database to return from UNIX *open*

Call UNIX *fstat* with parameter file_descriptor in HFR_OFS database; set stat_buf_ptr to returned stat structure and set Status equal to returned function value.

If Status equals NOERROR

Initialize appropriate HFR_Open_file database values

return Status

F.4.3 EFFECTS

HFR_Open_File Database structure is set:

Open_status - set to TRUE.

File_descriptor - contains UNIX file descriptor value.

File_size - contains the size of the file in bytes.

F.4.4 EXCEPTIONS

UNIX exceptions.

F.4.5 OUTPUTS

Status: NOERROR or EXCEPTION of the function.

F.4.6 DATABASE REFERENCED

HFR_Open_file Database.

F.4.7 INTERFACE: int hfr_open();

F.5 HFR_Read reads a block of bytes at a time from the file indicated in the HFR_Open_file Database and stores them in the Block_array. Passes back the Read_array with MAX_SEGMENT_SIZE number of bytes or less if EOF encountered. The logic of this function requires that MAX_BLOCK_SIZE be a multiple of MAX_SEGMENT_SIZE.

F.5.1 INPUTS

None.

F.5.2 PROCESSING

Local Variable:

int read_array_index = 0;

Initialize Status to NOERROR

If Block_bytes_read in the HFR_OFS database is equal to zero and More in the HFR_OFS database is equal to TRUE

Read MAX_BLOCK_SIZE of bytes from file indicated by file_descriptor in HFR_OFS database

Set Block_bytes_read in the HFR_OFS database equal to actual number of bytes read into block

Set Tot_bytes_read in the HFR_OFS database equal to Tot_bytes_read plus block_bytes_read
If EOF encountered
Set More in the HFR_OFS database equal to FALSE
While the block_index in the HFR_OFS database is less than block_bytes_read in the HFR_OFS database and the read_array_index is less than MAX_SEGMENT_SIZE
Assign read_array with byte from block_array in the HFR_OFS database
Increment read_array_index by one
Increment block_array_index in the HFR_OFS database by one
Set bytes_read equal to read_array_index plus one
If at end of block_array (block_index equals MAX_BLOCK_SIZE - 1)
Set block_index in the HFR_OFS database equal to zero
Set block_bytes_read in the HFR_OFS database equal to zero
return Status

F.5.3 EFFECTS

HFR_Open_file database is updated:
tot_bytes_read is updated if file read required.
block_array holds the most current bytes read from the file.
block_bytes_read contains the current number of bytes read into the block_array.
block_index is advanced by the number of Bytes_read or reset to zero if end of block reached.

F.5.4 EXCEPTIONS

UNIX exceptions.

F.5.5 OUTPUTS

Read_array: storage structure for passing the bytes read from the file.
Bytes_read: number of bytes contained in the read_array (this is always positive).
Status: NOERROR or EXCEPTION of the function.

F.5.6 DATABASE REFERENCED

HFR_Open_file Database

F.5.7 INTERFACE: int hfr_read(SEGMENT_type Read_array,
int &Bytes_read);

F.6 HFR_Close sets the open_status to FALSE in the HFR_Open_file database for the file that has been opened by the function **HFR_Open**.

F.6.1 INPUTS
None

F.6.2 PROCESSING
Initialize Status equal to NOERROR
If Open_status in the HFR_OFS database equals TRUE
    Call UNIX *close* with parameter file_descriptor in the HFR_OFS database
      and set Status equal to returned function value.
If Status equals NOERROR
    Set open_status in the HFR_OFS database equal to FALSE
return Status

F.6.3 EFFECTS
HFR_Open_file database:
    open_status is set to FALSE.

F.6.4 EXCEPTIONS
UNIX exceptions.

F.6.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

F.6.6 DATABASE REFERENCED
HFR_Open_file Database

F.6.7 INTERFACE: int hfr_close();

G. HIGH MODULATOR TRUNCATE MODULE

G.1 MODULE CONSTANTS:

int KBYTE = 1024
int BASE = 8 -- number of Kbytes to truncate each file
int MAX_KBYTE = PATHS * BASE -- maximum codes
  possible
int TRUNC_MENU_LINES = 4
int TRUNC_PROMPT_LINES = 1
int DEFAULT = 1 -- truncate menu choice for default paths
int USER_PATHS = 2 -- truncate menu choice for user defined
Default truncate file paths:
  PATHNAME_type TRUNC_PATH1 = "./t1"
  PATHNAME_type TRUNC_PATH2 = "./t2"
  PATHNAME_type TRUNC_PATH3 = "./t3"
  PATHNAME_type TRUNC_PATH4 = "./t4"

PATHNAME_type TRUNC_PATH5 = "./t5"
int START_TRUNC = 0 -- signals HMT_Truncate function to truncate the files to TRUNC_BASE size.
int TRUNC_BASE = 10 -- the minimum number of Kbytes the files will always be truncated. Thus, when truncate files are deleted the Low user will see a negative file system size change signaling the end.
int FILE_BASE = 8 -- maximum number of Kbytes each file can be truncated.

TYPES:
None.

VARIABLES:
MENU_LINE_type trunc_menu[TRUNC_MENU_LINES] = {
"TRUNCATE MENU",
"1. Use the default truncate file paths.",
"2. Enter the truncate file paths to use.",
"Enter your choice: "
};
MENU_LINE_type trunc_prompt = {
"Enter the truncate file paths, one per line."
};
HMT_Pathnames -- two dimensional character array of length MAX_PATH_LENGTH and depth of PATHS
PATHNAME_type HMT_Pathnames[PATHS];

G.2 DATABASES:
None.

G.3 HMT_Init initializes the paths to the files to be used for modulating the file system size in the HMT_Pathnames array to either the default pathnames or user defined pathnames.

G.3.1 INPUTS
None.

G.3.2 PROCESSING
Local Variables:
char Choice; -- variable for storing user's choice from trunc_menu
int Done = FALSE; -- loop control variable
Initialize Status to NOERROR
Loop while Done equals FALSE and Status equals NOERROR

Call Display with parameters trunc_menu and TRUNC_MENU_LINES and set Status equal to returned function value.
If Status equals NOERROR and Choice equals DEFAULT
Initialize HMT_Pathnames to default paths (TRUNC_PATH1 thru 5)
If Status equals NOERROR and Choice equals USER_PATHS
Call Display with parameters trunc_prompt and TRUNC_PROMPT_LINES; set Status equal to returned function value.
Loop for number of PATHS and while Status equals NOERROR
Call UNIX C-library *gets*; set HMT_Pathnames equal to returned character string and Status equal to returned function value.
If Status equals NOERROR and value of Choice is valid
Call HMT_Truncate with parameter START_TRUNC and set Status equal to returned function value.
If Status equals NOERROR
Set Done equal to TRUE
If Status equals EXCEPTION and UNIX *errno* equals invalid path exception
Call Display with parameters path_error and ERROR_LINES; set Status equal to returned function value.
If value of Choice is invalid
Call Display with parameters menu_error and ERROR_LINES and set Status equal to returned function value.
return Status

G.3.3 EFFECTS
None.

G.3.4 EXCEPTIONS
UNIX exceptions.

G.3.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

G.3.6 DATABASE REFERENCED
None.

G.3.7 INTERFACE: int hmt_init();

G.4 HMT_Truncate truncates the files in HMT_Pathnames to the specified number of Kbytes.

G.4.1 INPUTS

Kbytes_to_trunc: number of bytes the files are to be truncated.

G.4.2 PROCESSING

Local Variables:

int Base_files; -- number of files to be truncated to BASE value

Initialize Status to NOERROR

If Kbytes_to_trunc is greater than MAX_KBYTES

Set Status equal to EXCEPTION

If Kbytes_to_trunc is equal to EOF_SIGNAL

Call UNIX *truncate* for each of the files with parameters: path from Pathnames_array and length equal to zero

If Status equals NOERROR

***Files 1 & 2 will always be truncated to TRUNC_BASE divided by two plus any other bytes determined below

Set Base_files equal to integer result of Kbytes_to_trunc divided by BASE

Truncate Base_files number files to BASE

Call UNIX *truncate* for each of the files to be truncated to BASE with parameters: path from Pathnames_array and length equal to BASE

Truncate the next file to the number of Kbytes resulting from Kbytes_to_trunc modula BASE

Call UNIX *truncate* with parameters: next path from Pathnames_array and length equal to result of Kbytes_to_trunc modula BASE

Truncate any remaining files to zero

Call UNIX *truncate* for number of files to be truncated to zero with parameters: path from Pathnames_array and length equal to zero

return Status

G.4.3 EFFECTS

None.

G.4.4 EXCEPTIONS

UNIX exceptions.

G.4.5 OUTPUTS

Status: NOERROR or EXCEPTION of the function.

G.4.6 DATABASE REFERENCED

None.

G.4.7 INTERFACE: int hmt_truncate(int &Kbytes_to_trunc);

G.5 HMW_Delete deletes all the files used for truncating.

G.5.1 INPUTS
None.

G.5.2 PROCESSING
Initialize Status to NOERROR
Loop incrementing through HMT_Pathnames array while Status equals NOERROR
    Call UNIX *unlink* with path as the parameter and set Status equal to returned function value.
return Status

G.5.3 EFFECTS
None.

G.5.4 EXCEPTIONS
UNIX exceptions.

G.5.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

G.5.6 DATABASE REFERENCED
None.

G.5.7 INTERFACE: int hmt_delete();

H. LOW CONFIGURATION SYSTEM PARAMETERS

H.1 MODULE CONSTANTS:

None.

TYPES:

None.

VARIABLES:

None.

I. LOW MAIN MODULE

I.1 MODULE CONSTANTS:

int MAIN_MENU_LINES = 4
char PROCESS = '1'

TYPES:

None.

VARIABLES:

```
MENU_LINE_type  main_menu[MAIN_MENU_LINES] = {
    "MAIN MENU",
    "1. Receive and process a codes.",
    "9. Quit, exit to system prompt.",
    "Enter your choice: "
};
```

I.2 DATABASES:

None.

I.3 Low_main loops calling **LMC_Covert_init** and **LMC_run** until the user is done receiving and processing codes.

I.3.1 INPUTS

None.

I.3.2 PROCESSING

Local Variables:

char Choice = PROCESS; -- variable to store user's main_menu choice
char Covert_choice; -- variable to store user's covert_menu choice

Initialize Status to NOERROR
Loop while Choice is not equal to QUIT

Call Display with parameters main_menu and MAIN_MENU_LINES and set Status equal to returned function value.
If Status equals NOERROR
Call UNIX C-library *getchar*, set Choice equal to returned character and set Status equal to returned function value.
If Status equals NOERROR and Choice equals PROCESS
Call LMC_Covert_init, set Covert_choice equal to returned character and set Status equal to returned function value.
If Status equals NOERROR and Covert_choice equals QUIT
Set Choice equal to QUIT
If Status equals NOERROR and Choice equals PROCESS
Call LMC_Covert_run with no parameters and set Status equal to returned function value.
If Status equals NOERROR and value of Choice is invalid
Call Display with parameters menu_error and ERROR_LINES and set Status equal to returned function value.
return Status

I.3.3 EFFECTS
None.

I.3.4 EXCEPTIONS
UNIX exceptions.

I.3.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

I.3.6 DATABASE REFERENCED
None.

I.3.7 INTERFACE: int low_main();

J. LOW_MODULATOR_CONTROLLER MODULE

J.1 MODULE CONSTANTS:

int COVERT_MENU_LINES = 4
int THEORY_MENU_LINES = 4
char PROBABILITY = '1'
char MODULATE = '1'
char UNKNOWN_CHAR = '?'

PROB_DIST -- Probability distribution codes structure

Alpha_codes -- character array containing the lowercase alphabetic characters in the appropriate probability distribution codes position..

Num_codes -- character array containing the numeric characters 0 thru 9 in the appropriate probability distribution codes position.

```
struct Prob_dist{
    char    alpha_codes[26] = {
            'e','t','a','o','i','n','s','r','h','l','d','c','u',
            'm','f','p','g','w','y','b','v','k','x','j','q','z'};
    char    num_codes[10] = {
            '0','1','2','3','4','5','6','7','8','9'};
};
```

TYPES:

LMC_CDB_type -- LMC_Covert Database structure

Covert_method - integer value indicating covert method to use.

Info_theory_method - integer value indicating info theory method to use.

```
typedef  int LMC_CDB_type[2];
```

VARIABLES:

```
MENU_LINE_type covert_menu[COVERT_MENU_LINES] =
{
    "COVERT METHOD MENU",
    "1. Read file system modulation.",
    "9. Quit, return to Main menu.",
    "Enter your choice: "
};
MENU_LINE_type theory_menu[THEORY_MENU_LINES] =
{
    "INFORMATION THEORY METHOD MENU",
    "1. Probability distribution of characters method.",
    "9. Quit, return to Covert Method menu.",
    "Enter your choice: "
};
LMC_CDB_type  CDB; -- Covert database
```

J.2 DATABASES:

LMC_Covert Database.

J.3 LMC_Covert_init sets the Covert_method in the LMC_Covert database equal to the user's choice. Calls other module initialization functions based on the covert method chosen.

J.3.1 INPUTS

None.

J.3.2 PROCESSING

Local Variables:
    char Theory_choice; -- theory menu choice.
Initialize Status to NOERROR
Set Covert_choice equal to MODULATE
Loop while Covert_choice is not equal to QUIT
    Call Display with parameters Covert_menu and COVERT_MENU_LINES
      and set Status equal to returned function value.
    If Status equals NOERROR
        Call UNIX C-library *getchar*, set Covert_choice equal to returned
          character and set Status equal to returned function value.
    If Status equals NOERROR and Covert_choice equals MODULATE
        Set Covert_method in LMC_Covert database equal to MODULATE
        Call LMC_Info_theory_init, set Theory_choice equal to returned
          character and set Status equal to returned function value.
    If Status equals NOERROR and Theory_choice equals QUIT
        Set Covert_choice equal to QUIT
    If Status equals NOERROR and Covert_choice equals MODULATE
        Call LFW_Init with no parameters and set Status equal to returned
          function value.
    If Status equals NOERROR and Covert_choice equals MODULATE
        Call LMR_Init with no parameters and set Status equal to returned
          function value.
        Set Covert_choice equal to QUIT
  If Status equals NOERROR and value of Covert_choice is invalid
      Call Display with parameters menu_error and ERROR_LINES
return Status;

J.3.3 EFFECTS

LMC_Covert database is initialized to contain covert method to use.

J.3.4 EXCEPTIONS

UNIX exceptions.

J.3.5 OUTPUTS
Covert_choice: character value containing user's Covert_menu choice.
Status: NOERROR or EXCEPTION of the function.

J.3.6 DATABASE REFERENCED
LMC_Covert database.

J.3.7 INTERFACE: int lmc_covert_init(char Covert_method);

J.4 LMC_Info_theory_init sets the Info_theory_method in the LMC_Covert database equal to the user's choice. Initializes needed structures based on the information theory method chosen.

J.4.1 INPUTS
None.

J.4.2 PROCESSING
Initialize Status to NOERROR
Set Theory_choice equal to MODULATE
Loop while Theory_choice is not equal to QUIT
    Call Display with parameters Theory_menu and THEORY_MENU_LINES
      and set Status equal to returned function value.
    If Status equals NOERROR
        Call UNIX C-library *getchar*, set Theory_choice equal to returned
          character and set Status equal to returned function value.
    If Status equals NOERROR and Theory_choice equals PROBABILITY
        Set Theory_method in LMC_Covert database equal to PROBABILITY
    If Status equals NOERROR and value of Theory_choice is invalid
      Call Display with parameters menu_error and ERROR_LINES
return Status;

J.4.3 EFFECTS
LMC_Covert database is initialized to contain info theory method to use.

J.4.4 EXCEPTIONS
UNIX exceptions.

J.4.5 OUTPUTS
Theory_choice: character value containing user's Theory_menu choice.
Status: NOERROR or EXCEPTION of the function.

J.4.6 DATABASE REFERENCED
LMC_Covert database.

J.4.7 INTERFACE: int lmc_info_theory_init(char Theory_choice);

J.5 LMC_Covert_run calls the appropriate covert channel method based on the value of Covert_method in the LMC_Covert database.

J.5.1 INPUTS
None.

J.5.2 PROCESSING
Initialize Status equal to NOERROR
If Staus equals NOERROR and Covert_method in LMC_Covert database equals
MODULATE
Call LMC_Modulator_controller with no parameters and set Status equal to returned function value.
return Status;

J.5.3 EFFECTS
None.

J.5.4 EXCEPTIONS
UNIX exceptions.

J.5.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

J.5.6 DATABASE REFERENCED
LMC_Covert database.

J.5.7 INTERFACE: int lmc_covert_run();

J.6 LMC_Modulator_controller calls **LMR_Read, LMC_Info_theory** and **LFW_Write** repeatedly reading the number of free system blocks from the file system disk that the LMR_Pathname file resides on until a negative block size change is received from **LMR_Read** signalling the end of transmission. It then calls **LFW_Close** to close the write file that was opened.

J.6.1 INPUTS
None.

J.6.2 PROCESSING
Local Variables:
int Temp_bytes; -- holds value of base minus kbytes read.
int Kbytes_read; -- value of return parameter from LMR_Read.
int Base_kbytes; -- initial value of return parameter from LMR_Read.

int Bytes_to_write; -- number of bytes in Write and Code arrays.
SEGMENT_type Write_array; -- storage structure for returned character array from LMC_Info_theory.
CODE_type Code_array; -- storage structure for returned integers from LMR_Read.
Initialize Status to NOERROR
Call LMR_Read; set Base_kbytes equal to returned integer value and Status equal to returned function value.
Loop while Status equals NOERROR and Temp_bytes is greater than EOF_SIGNAL
Set Bytes_to_write equal to zero
Loop while Status equals NOERROR, Kbytes_read is greater than zero and Bytes_to_write is less than MAX_SEGMENT_SIZE
Call LMR_Read; set Kbytes_read equal to returned integer value and Status equal to returned function value.
Set Bytes_to_write position in Code_array equal to Base_kbytes minus Kbytes_read
Increment Bytes_to_write by one
If Status equals NOERROR
Call LMC_Info_theory with Code_array and Bytes_to_write; set Write_array equal to returned character array and Status equal to returned function value.
If Status equals NOERROR
Call LFW_Write with Write_array and Bytes_to_write; set Status equal to returned function value.
Call LFW_Close with no parameters and set Status equal to returned function value.
return Status

J.6.3 EFFECTS
None.

J.6.4 EXCEPTIONS
UNIX exceptions.

J.6.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

J.6.6 DATABASE REFERENCED
None.

J.6.7 INTERFACE: int lmc_modulator_controller( );

J.7 LMC_Info_theory calls the appropriate decoding function based on the Info_theory_method value in the LMC_Covert database.

J.7.1 INPUTS

Code_array: storage structure for passing the codes read from the file system.
Bytes_to_write: number of bytes contained in the Code and Write arrays (this is always positive).

J.7.2 PROCESSING

Initialize Status to NOERROR
If Info_theory_method in the LMC_Covert database is equal to PROBABILITY
    Call LMC_Char_prob_dist with parameters Code_array and
      Bytes_to_write; set Write_array equal to returnen character array and set
      Status equal to returned function value.
return Status

J.7.3 EFFECTS

None.

J.7.4 EXCEPTIONS

UNIX exceptions.

J.7.5 OUTPUTS

Write_array: storage structure for passing the converted character codes.
Status: NOERROR or EXCEPTION of the function.

J.7.6 DATABASE REFERENCED

LMC_Covert database.

J.7.7 INTERFACE:

```
int lmc_info_theory(CODE_type *Code_array,
                    int &Bytes_to_write,
                    SEGMENT_type *Write_array);
```

J.8 LMC_Char_prob_dist converts (decodes) the integer codes in the Code_array to characters and stores the characters in the Write_array.

J.8.1 INPUTS

Code_array: storage structure for passing the converted character codes.
Bytes_to_write: number of bytes contained in the Code and Write arrays (this is always positive).

J.8.2 PROCESSING

Initialize Status to NOERROR
Loop incrementing through Code_array for number of Bytes_to_write
    If code is less than 26 then convert using Alphabetic_code_array
        Store decoded character in Write_array
    If code is greater than 25 and less than 35 then convert using
      Numeric_code_array
        Store decoded character in Write_array
    If code is greater than 35
        Store UNKNOWN_CHAR character in Write_array
return Status

J.8.3 EFFECTS

None.

J.8.4 EXCEPTIONS

None.

J.8.5 OUTPUTS

Write_array: storage structure for passing the characters.
Status: NOERROR or EXCEPTION of the function.

J.8.6 DATABASE REFERENCED

None.

J.8.7 INTERFACE:

```
int lmc_char_prob_dist(
                        CODE_type Code_array,
                        int &Bytes_to_write,
                        SEGMENT_type Write_array);
```

K. LOW FILE WRITER MODULE

K.1 MODULE CONSTANTS:

```
int WRITE_PROMPT_LINES = 1
```

TYPES:

```
LFW_OFS_type -- LFW_Open_file Database structure
    typedef  struct {
        int             open_status; - boolean value indicating
                            open status of file
        fsid_t          file_descriptor; - integer assigned by
                            the system when a file is opened
    }LFW_OFS_type;
```

VARIABLES:

```
MENU_LINE_type  write_prompt = {
  "Enter the path to the file you want to write to: "
};
LFW_OFS_type LFW_OFS;
LFW_OFS_type *LFW_OFS_ptr = LFW_OFS;
PATHNAME_type  LFW_Pathname;
```

K.2 DATABASES:

LFW_Open_file Database.

K.3 LFW_Init initializes the LFW_Open_file Database structure. Gets the pathname of the file to write to from the user and calls **LFW_Open** to open or create the file.

K.3.1 INPUTS

None.

K.3.2 PROCESSING

Local Variables:

int Done = FALSE; -- loop control variable

Initialize Status to NOERROR

Loop while Done equals FALSE

Call Display with parameters write_prompt and WRITE_PROMPT_LINES; set Status equal to returned function value.

If Status equals NOERROR

Call UNIX C-library *gets*, set LFW_Pathname equal to returned character array and set Status equal to returned function value.

If Status equals NOERROR

Initialize LFW_Open_file database

Call LFW_Open with no parameters and set Status equal to returned function value.

If Status equals NOERROR

Set Done equal to TRUE

If Status equals EXCEPTION and UNIX *errno* equals invalid path exception

Call Display with parameters path_error and ERROR_LINES; set Status equal to returned function value.

return Status

K.3.3 EFFECTS

LFW_Open_file database is initialized:

open_status is set to FALSE.

K.3.4 EXCEPTIONS
UNIX exceptions.

K.3.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

K.3.6 DATABASE REFERENCED
LFW_Open_file database

K.3.7 INTERFACE: int lfw_init();

K.4 LFW_Open opens or creates a file for writing and sets the appropriate values in the LFW_Open_file database structure.

K.4.1 INPUTS
LFW_Pathname: storage structure for passing the path to the file to open or create.

K.4.2 PROCESSING
Initialize Status to NOERROR
Call UNIX *open* with parameter LFW_Pathname and set Status equal to returned function value.
If Status equals NOERROR
    Set file_descriptor in LFW_OFS database equal to Status
If Status equals NOERROR
    Initialize appropriate LFW_Open_file database values
return Status

K.4.3 EFFECTS
LFW_Open_File Database structure is set:
    Open_status - set to TRUE.
    File_descriptor - contains UNIX file descriptor value.

K.4.4 EXCEPTIONS
UNIX exceptions.

K.4.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

K.4.6 DATABASE REFERENCED
LFW_Open_file Database

K.4.7 INTERFACE: int lfw_open(PATHNAME_type LFW_Pathname);

K.5 LFW_Write writes the characters contained in the Write_array to the file indicated in the LFW_Open_file Database.

K.5.1 INPUTS

Write_array: storage structure for passing the characters to be written.

Bytes_to_write: number of bytes contained in the Write_array (this is always positive).

K.5.2 PROCESSING

Initialize Status to NOERROR

Call UNIX *write* with file_descriptor in LFW_Open_file database, pointer to Write_array and Bytes_to_write; set Status equal to returned function value.

return Status

K.5.3 EFFECTS

None.

K.5.4 EXCEPTIONS

UNIX exceptions.

K.5.5 OUTPUTS

Status: NOERROR or EXCEPTION of the function.

K.5.6 DATABASE REFERENCED

LFW_Open_file Database

K.5.7 INTERFACE: int lfr_write(SEGMENT_type Write_array,
int &Bytes_to_write);

K.6 LFW_Close sets the open_status to FALSE in the LFW_Open_file database for the file that has been opened by the function **LFW_Open**.

K.6.1 INPUTS

None

K.6.2 PROCESSING

Initialize Status equal to NOERROR

If Open_status in the LFW_Open_file database equals TRUE

Call UNIX *close* with parameter file_descriptor and set Status equal to returned function value.

If Status equals NOERROR
    Set open_status in the LFW_Open_file database equal to FALSE
return Status

K.6.3 EFFECTS
LFW_Open_File Database has Open_status set to FALSE.

K.6.4 EXCEPTIONS
UNIX exceptions.

K.6.5 OUTPUTS
Status: NOERROR or EXCEPTION of the function.

K.6.6 DATABASE REFERENCED
LFW_Open_file Database

K.6.7 INTERFACE: int lfw_close();

L. LOW MODULATOR READER MODULE

L.1 MODULE CONSTANTS:

int READ_PROMPT_LINES = 1

TYPES:
None.

VARIABLES:

```
MENU_LINE_type read_prompt = {
  "Enter the path to the file on the file system you want to read:"
};
LMR_Pathname -- character array of length
    MAX_PATH_LENGTH
      PATHNAME_type  *LMR_Pathname;
```

L.2 DATABASES:

None.

L.3 LMR_Init initializes the LMR_Pathname array to the path of a file on the file system that the user wants to monitor the file system size modulation.

L.3.1 INPUTS
None.

L.3.2 PROCESSING

Local Variables:

struct stat stat_buffer -- structure for holding return values from UNIX *stat*
stat *stat_buf_ptr = stat_buffer;
int Done = FALSE; -- loop control variable

Initialize Status to NOERROR
Loop while Done equals FALSE
Call Display with parameters read_prompt and READ_PROMPT_LINES; set Status equal to returned function value.
If Status equals NOERROR
Call UNIX C-library *gets*, set LMR_Pathname equal to returned character array and set Status equal to returned function value.
If Status equals NOERROR
Call UNIX *stat* with parameter LMR_Pathname; set stat_buf_ptr to returned stat structure and set Status equal to returned function value.
If Status equals NOERROR
Set Done equal to TRUE
If Status equals EXCEPTION and UNIX *errno* equals invalid path exception
Call Display with parameters path_error and ERROR_LINES; set Status equal to returned function value.
return Status

L.3.3 EFFECTS

None.

L.3.4 EXCEPTIONS

UNIX exceptions.

L.3.5 OUTPUTS

Status: NOERROR or EXCEPTION of the function.

L.3.6 DATABASE REFERENCED

None.

L.3.7 INTERFACE: int LMT_init();

L.4 LMR_Read determines and returns the number of free blocks on the file system refered to by LMR_pathname. The UNIX block size is 1024 bytes.

L.4.1 INPUTS

Kbytes_read: number of free blocks on the file system.

L.4.2 PROCESSING

Local Variables:

struct statfs buf -- structure for holding return values from UNIX *fstatfs*
statfs *buf_ptr = buf;

Initialize Status to NOERROR

Call UNIX *statfs* with parameters LMR_Pathname; set buf_ptr equal to returned *statfs* structure and set Status equal to returned function value.

If Status equals NOERROR

Set Kbytes_read equal to *f_bfree* in *statfs* struct (Kbytes_read = buf_ptr -> f_bfree)

return Status

L.4.3 EFFECTS

None.

L.4.4 EXCEPTIONS

UNIX exceptions.

L.4.5 OUTPUTS

Status: NOERROR or EXCEPTION of the function.

L.4.6 DATABASE REFERENCED

None.

L.4.7 INTERFACE: int lmr_read(int &Kbytes_read);

LIST OF REFERENCES

1. National Computer Security Center. *DoD Trusted Computer System Evaluation Criteria*, Department of Defense, DoD 5200.28-STD, December 1985.

2. Gasser, Morrie. *Building a Secure Computer System*, New York: Van Nostrand Reinhold, 1988.

3. Amoroso, Edward G. *Fundamentals of Computer Security Technology*, Prentice-Hall, Inc., 1994.

4. Bell, D. E., and LaPadula, L. J. *Secure Computer Systems: Mathematical Foundations*, ESD-TR-73-278, Vol. I, Mitre Corporation, 1973.

5. Graham, G. S., and Denning, P. J. "Protection--Principles and Practice." *Proc. Spring Jt. Computer Conf*, Vol. 40. Montvale, N. J.: AFIPS Press (1972): 417-429.

6. Biba, K. *Integrity Considerations for Secure Computer Systems*, MTR-3153, Mitre Corporation, 1975.

7. Lampson, Butler. "A Note on the Confinement Problem." *Communications of the ACM*, Vol.16, No.10 (October 1973): 613-615.

8. SUN Microsystems Federal, Inc. *Trusted Facility Manual for Trusted Solaris 1.1*. Vol. I & II, Revision A, February 1994.

9. van Tilborg, Henk C. A. *An Introduction to Cryptology*, Boston: Kluwer Academic Publishers.

10. Booch, G., and Bryan, D. *Software Engineering with ADA*, 3rd ed. Redwood City, California: Benjamin/Cummings Publishing Company, Inc., 1994.

11. Parnas, D. L. "On the Criteria To Be Used in Decomposing Systems into Modules." *Communications of the ACM* Vol.15, No.12 (December 1972): 1053-1058.

12. Berzins, V., and Luqi. *Software Engineering with Abstractions*. New York: Addison-Wesley Publishing Company, 1991.

13. Yourdon, E. *Modern Structured Analysis*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1989.

14. National Computer Security Center. *A Guide to Understanding Audit in Trusted Systems*, NCSC-TG-001 Ver. 2, 1 June 1988.

15. Gligor, Virgil D. *Guidelines for Trusted Facility Management and Audit*. University of Maryland, 1985.

16. Proctor, P. "Audit Reduction and Misuse Detection in Heterogeneous Environments: Framework and Application." *IEEE Computer Security Applications Conference.* Orlando, Florida. (December 1994) : 117-125.

17. Fisch, E. A., White, G. B., and Pooch, U. W. "The Design of an Audit Trail Analysis Tool." *IEEE Computer Security Applications Conference.* Orlando, Florida. (December 1994) : 126-132.

18. National Computer Security Center. *A Guide to Understanding Covert Channel Analysis of Trusted Systems*, NCSC-TG-030 Ver. 1, November 1993.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library 2
   Code 052
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr Cynthia E. Irvine, Code CS/IC 6
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Dr Timothy J. Shimeall, Code CS/SM 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Sun Microsystems Federal 2
   ATTN: Don Adams
   2550 Garcia Avenue
   Mountain View, CA 94043
   Mail Stop 06-07

7. Prof. Roger Stemp, Code CS/Sp 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

8. Albert Wong, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

9. Cpt Ronald J. DeJong 2
1250 Monroe Blvd
South Haven, MI 49090